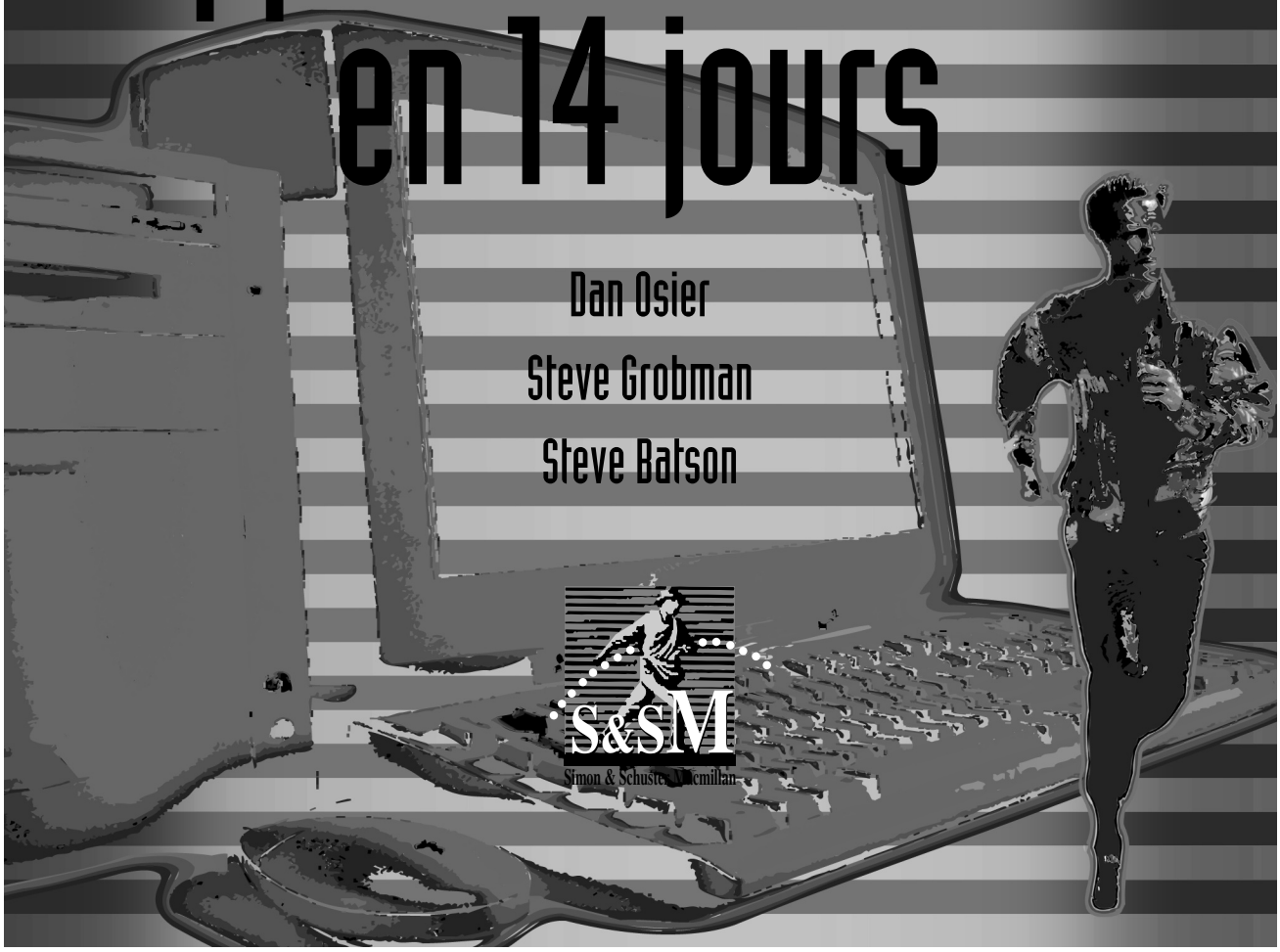
LE PROGRAMMEUR

Apprenez DELPHI 3 en 14 jours

Dan Osier

Steve Grobman

Steve Batson



S&SM

Simon & Schuster Macmillan

Simon & Schuster Macmillan (France) a apporté le plus grand soin à la réalisation de ce livre afin de vous fournir une information complète et fiable. Cependant, Simon & Schuster Macmillan (France) n'assume de responsabilités, ni pour son utilisation, ni pour les contrefaçons de brevets ou atteintes aux droits de tierces personnes qui pourraient résulter de cette utilisation.

Les exemples ou les programmes présents dans cet ouvrage sont fournis pour illustrer les descriptions théoriques. Ils ne sont en aucun cas destinés à une utilisation commerciale ou professionnelle.

Tous les noms de produits ou marques cités dans ce livre sont des marques déposées par leurs propriétaires respectifs.

Publié par Simon & Schuster Macmillan (France)
19, rue Michel Le Comte
75003 PARIS
Tél : 01 44 54 51 10
Mise en page : Andassa

Titre original : Teach Yourself Delphi 3 in 14 days
Traduit de l'américain par : Pierre Couzy et Yann Schwartz
ISBN original : 0-672-31074-0

SAM'S NET est une marque de Macmillan Computer Publishing USA
201 West 3rd street
Indianapolis, Indiana 46290. USA

Sommaire

1

Présentation de Delphi 3

Lors de cette première journée, nous allons vous présenter les caractéristiques de Delphi 3. Nous tracerons un historique succinct des outils de développement, et nous verrons les éléments qui ont présidé à la création de cet outil puissant et sans équivalent qu'est Delphi. Après une brève initiation à Delphi, nous écrirons une courte application pour nous entraîner. Delphi 3 est une sorte de "Visual Pascal", et bien plus. Delphi 3 vous permet de tirer partie à la fois des solides fondations du Pascal Objet de Borland et des fonctionnalités de construction d'applications visuelles que proposent des produits tels que Visual Basic. Delphi présente de nombreux avantages par rapport à ses concurrents, et peut simplifier grandement le travail du programmeur. Nous terminerons cette journée en passant en revue l'EDI (Environnement de Développement Intégré).

Delphi pour une programmation RADieuse

Nouveau

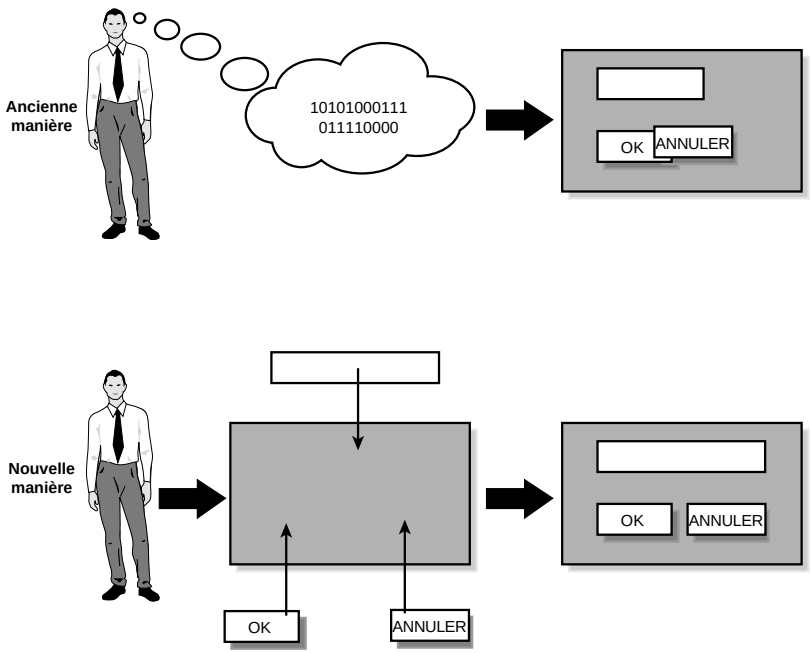
*RAD signifie Développement rapide d'application (*Rapid Application Development*). Ce terme décrit la nouvelle génération d'environnements de développement logiciel. Dans un environnement RAD, les programmeurs utilisent des outils plus intuitifs et visuels. Il est difficile de regarder un bout de code qui crée une fenêtre et de la visualiser, mais le RAD permet de créer la fenêtre en quelques clics.*

Parmi ces nouvelles interfaces, plus simples et plus visuelles, la première à faire son apparition fut Visual Basic (ou VB). VB a mis la programmation — qui était jusque-là une pratique mystique réservée à quelques gourous — à la portée des simples mortels. Cette nouvelle interface permit, comme illustré dans la Figure 1.1, au développeur de logiciel de construire "visuellement" l'interface utilisateur à l'aide de la souris, au lieu de le faire en codant, puis en compilant pour vérifier le résultat. L'ancienne méthode revenait à créer les moules pour un nouvelle voiture avant même de créer une maquette reproduisant les formes du bolide....

Avant de poursuivre, il convient de préciser que rien ne remplace de solides bases en programmation. C'est dans cette optique que vous trouverez dans cet ouvrage des sections consacrées à la *programmation orientée objet* (Jour 7), et à la conception d'une *GUI* (*Interface graphique utilisateur*) et de bases de données (Jour 6), qui représentent de bonnes bases pour devenir un développeur de logiciel chevronné.

Si VB rencontra un grand succès et permit d'ouvrir au grand public les portes de la programmation, il n'était pas sans défaut. Le langage lui-même n'encourageait pas de bonnes méthodes de programmation. VB ne contenait pas les mécanismes qui favorisent un code clair, compact et très structuré. Enfin, VB n'avait pas la rigueur d'un langage orienté objet (nous reviendrons en détail à ce sujet au Jour 4). Pire encore, VB encourageait la "mauvaise" programmation, en permettant au développeur de bricoler une solution rapide, contournant les techniques, parfois contraignantes, mais qui permettent de construire efficacement un programme. Les trois auteurs de ce livre sont de bons programmeurs VB (entre autres), et nous avons pu expérimenter les deux manières de faire (VB et programmation objet Delphi), et c'est en connaissance de cause que nous avons choisi Delphi.

Figure 1.1
L'ancienne approche et la nouvelle.



Delphi marque une évolution dans les environnements de développement RAD. Il corrige grand nombre des défauts de VB, sans en ajouter trop. Nous allons décrire les points forts et les points faibles — qui ne sont pas si nombreux — de Delphi.

Delphi, un Visual Basic sous stéroïdes

Pour lire ce livre, il n'est pas nécessaire d'avoir l'expérience d'un langage de programmation, mais si vous avez déjà programmé en VB, vous aurez l'impression que Delphi est un VB dopé. Les développeurs de Delphi ont créé un outil qui peut, de prime abord, rappeler l'environnement de VB 4, mais Delphi est en fait bien différent (et bien meilleur). L'EDI (Environnement de Développement Intégré) est appelé en double-cliquant sur l'icône Delphi. Même si visuellement, Delphi 3 peut rappeler VB, la différence majeure se cache "derrière" l'EDI.

En termes de développement, VB favorise la rapidité au détriment de la qualité structurelle. En revanche, Delphi utilise comme fondation le Pascal objet. Le compilateur Pascal de Borland (apparu pour la première fois avec Turbo Pascal 1.0) fut dès l'origine l'un des compilateurs les plus rapides du marché. Borland a ajouté des extensions objet au langage pour faciliter les bonnes habitudes de programmation et augmenter l'efficacité du code (afin d'obtenir plus en moins de lignes). Le Pascal Objet est un vrai langage objet, assisté d'un compilateur qui a fait ses preuves.

Avantages de Delphi

Delphi apporte une grande souplesse au développeur. Lorsque Delphi génère un fichier .exe, il s'agit d'un vrai exécutable. Aucun autre fichier n'est nécessaire pour l'exécution. Vous obtenez donc une application plus propre, et plus facile à distribuer et à maintenir. Vous n'avez à distribuer qu'un seul fichier, sans dépendre de DLL ou d'autres fichiers.

Pour les entreprises soucieuses de l'établissement de normes et de standards, Delphi est également utile. Supposons que vous écriviez des applications Delphi dans une entreprise de 5000 employés. Chaque fois que vous devez déployer une nouvelle application, vous devez envoyer un fichier de 1Mo à tous les utilisateurs. Ceci peut vite encombrer le réseau. Fort heureusement, Delphi vous permet de regrouper les composants standard dans un paquet. Une fois ce paquet installé sur les machines, vous pouvez l'utiliser pour toutes les nouvelles applications que vous déploierez. Dès lors, il vous suffit d'envoyer un fichier exécutable de 200 ko au lieu de 1 Mo.

Cette technique est l'une des nouvelles fonctionnalités de Delphi 3, et permet de minimiser la taille des applicatifs transmis à chaque machine dès lors qu'un paquet standard a été installé. Vous en apprendrez plus à ce sujet dans le Jour 5, "Applications, Fichiers et Gestionnaire de Projets".

Delphi vous offre donc un compilateur optimisé qui vous donnera une application rapide sans qu'il soit nécessaire de fournir plus d'efforts pour optimiser le programme qu'il n'en avait fallu pour l'écrire.

Les différences entre Delphi 3 et Delphi 2

Bien que l'EDI de Delphi 3 semble inchangé, on compte de nombreuses différences cachées sous le capot. Les principales améliorations sont les suivantes :

- Architecture des bases de données et connectivité. L'architecture des bases de données a été complètement revue afin de suivre une approche multi-liaison plutôt que la traditionnelle conception Client/Serveur. Vous pouvez ainsi créer des applications client extrêmement légères. Le support natif des bases de données Access a enfin été intégré, afin de permettre une transition aisée des applications VB sous Delphi 3. Nous aborderons ce sujet plus longuement au onzième jour.
- Contrôles ActiveX. Vous pouvez créer vos propres contrôles ActiveX, ou utiliser des contrôles déjà écrits, dans vos applications Delphi. Vous apprendrez au Jour 13 à écrire des composants VCL et ActiveX.
- Applications Web. Vous pouvez créer des applications client ou serveur, dans un environnement Web. Ceci donne à Delphi 3 de nombreux atouts dans la course à Intranet. Ce sujet est abordé lors de la quatorzième journée.

- Paquets. Vous pouvez choisir d'inclure les librairies d'exécution, soit dans l'exécutable, soit dans une DLL externe (liaison statique ou dynamique). Les développeurs indépendants préféreront sans doute la première solution, la seconde étant plus adaptée au développements en interne.
- ActiveForm. Vous pouvez exécuter une application Delphi 3 complète sous la forme d'un contrôle ActiveX. Vous pouvez littéralement exécuter votre application dans une page Web. Vous en apprendrez plus à ce sujet lors des Jours 13 et 14.
- Améliorations du débogueur et de l'EDI. La plupart des routines graphiques, ainsi que le travail sur les canvas, sont maintenant isolés des problèmes de threads.

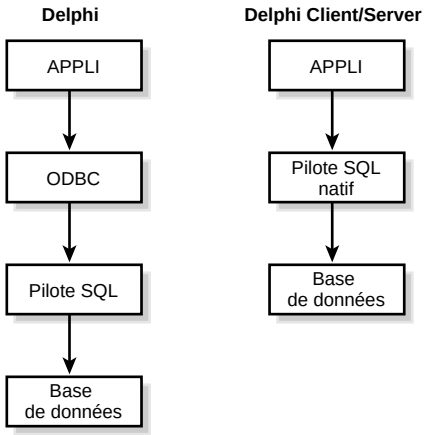
Info

En plus de la technologie ActiveX, Delphi 3 vous permet de créer des composants Delphi natifs que vous pouvez réutiliser dans tous vos projets. Alors qu'avec les contrôles ActiveX, il faut fournir le composant ActiveX avec votre application, il n'en va pas de même pour les composants natifs. Vous apprendrez à écrire vos propres composants visuels au Jour 13.

Qu'est-ce que Delphi Client/Serveur ?

Il existe trois versions de Delphi : Delphi Desktop, Delphi Développeur et Delphi Client/Serveur (que nous appellerons CS). A chacune de ces versions correspondent différents niveaux de connectivité avec le monde extérieur. Delphi Desktop permet de se connecter à dBASE, Paradox et Access au moyen du Borland Database Engine (Moteur de base de données Borland). La version Développeur permet la connectivité ODBC (pour connecter toute source de données au moyen d'un pilote ODBC), et la version CS propose des liens SQL, vous offrant ainsi des pilotes 32 bits ultra-rapides permettant de se connecter à des bases de données SQL server telles que SyBase et Oracle, comme indiqué dans la Figure 1.2.

Figure 1.2
L'édition CS permet un accès en mode natif aux bases de données.



La version CS a été conçue comme un concurrent des outils de développement d'application client/serveur, dont PowerBuilder est le chef de file. Ce genre d'outils est utilisé dans un cadre professionnel pour développer des applications Windows permettant d'accéder à des bases de données en réseau.

Composant Visuel

Delphi possède en outre sa propre version native d'un contrôle ActiveX, appelée Visual Component (Composant Visuel). Un VC est écrit en Delphi et s'ajoute à la Visual Component Library (Bibliothèque de Composants Visuels ou VCL). Cette VCL est un catalogue de VC qui peuvent être utilisés par le développeur pour créer des applications Delphi. Tous les composants de la VCL sont affichés sur la barre d'outils, permettant ainsi un accès facile. Vous en apprendrez plus à ce sujet au Jour 8.

Variables et constantes

Pour ceux d'entre vous qui n'ont jamais programmé, certains concepts peuvent vous être étrangers. En programmation, une *constante* est... constante. Si vous avez besoin d'utiliser le taux de TVA dans votre programme, vous pouvez entrer 0.206 (20,6 %) dans tous les formulaires de votre application. Si le taux de TVA change, il vous faudra alors revenir à vos formulaires et modifier chacun des 0.206 pour les transformer en 0.226, par exemple. Ceci constitue une monstrueuse perte de temps, avec le risque de commettre une faute de frappe en recopiant quarante fois le même taux.

C'est ici que les constantes interviennent. Vous pouvez définir une constante, `TauxTVA` par exemple, égale à 0.206 et utiliser ce `TauxTVA` là où c'est nécessaire. Il est important de noter que les constantes ne peuvent être modifiées au cours de l'exécution du programme. Il ne serait donc pas possible de demander à l'utilisateur le nouveau taux de TVA, puis de placer la nouvelle valeur dans `TauxTVA`. `TauxTVA` doit être défini dans le code, et ne peut être modifié en cours d'exécution.

```
Const
TauxTVA = 0.206;
```

Le principe d'une constante est que le nom (`TauxTVA`) est un substitut direct de 0.206. Ainsi, chaque fois que vous devez utiliser 0.206 dans votre application, vous pouvez utiliser à la place le nom de la constante. La formule `PrixTTC := PrixHT * TauxTVA;` peut sembler étrange mais elle marche.

Cette ligne de code, `PrixTTC := PrixHT * TauxTVA;` nous amène au sujet suivant, les *variables*. Vous vous souvenez peut-être de vos cours de maths au collège, et de ces démoniaques formules du type y=mx+b. Souvenez-vous de votre prof disant : "C'est facile, il suffit de résoudre l'équation en y". Eh bien, sans le savoir vous utilisiez des variables. Une variable est un emplacement contenant une valeur. Alors que les constantes restent immuables lors de l'exécution du programme, une variable peut être modifiée à tout moment.

Si nous avions défini une variable `TVA` sous forme d'un nombre Single, cette variable `TVA` peut maintenant contenir un nombre `Single`. Le type de données `Single` est un nombre réel représentant un nombre entre 1.5×10^{-45} et 3.4×1038. Ce type est tout à fait adapté à nos besoins.

```
Var
TVA : Single;
```

La déclaration précédente indique que vous venez de créer une variable appelée `TVA`, variable de type `Single`. Maintenant, dans votre programme, vous pouvez demander le taux à l'utilisateur et placer ce taux dans la variable `TVA`.

```
TVA := {Placez ici le taux donné par l'utilisateur};
```

Le grand avantage d'une variable est qu'il est maintenant possible de modifier la valeur de `TVA` en cours d'exécution du programme. Les variables sont réutilisables, et bien plus souples à l'usage que les constantes.

Il existe une différence foncière entre une constante et une variable, dans leur fonction bien sûr, mais aussi dans la façon dont elles sont utilisées dans le compilateur. Comme une constante est un substitut pour une valeur, lorsque vous compilez votre application, Delphi se contente de remplacer tous les `TauxTVA` qu'il trouve par la valeur explicite 0.206. Ceci n'a lieu que dans l'exécutable généré par Delphi, et votre source n'est pas modifiée. Une variable est gérée d'une manière bien différente. Comme la valeur d'une variable peut être modifiée au cours de l'exécution d'un programme, l'application doit allouer un espace mémoire pour y stocker `TVA`, afin de se souvenir de sa valeur et de modifier éventuellement cette dernière en cas de changement en cours de programme. Nous reviendrons plus en détail sur les constantes et les variables lors de la deuxième journée.

Procédures et fonctions

Lorsque vous commencez à écrire des programmes, vous risquez d'être victime du syndrome de Von Neumann. John Von Neumann est le père de l'informatique séquentielle où, du début à la fin, tout se passe séquentiellement, de manière linéaire. Voyons comment il est possible de se démarquer de cette technique.

Vous avez écrit un programme qui affiche un message de bienvenue trois fois de suite à l'écran. Pour écrire ce programme, vous écrirez quelque chose qui ressemble au Listing 1.1. Il s'agit ici de pseudo-code, destiné principalement à vous donner une idée de ce qui se passe.

Le pseudo-code est une interprétation en français d'événements, qui simule ou imite le code. Nous l'utilisons dans le listing pour montrer la forme générale d'un programme sans pour autant employer le code Pascal objet véritable.

Listing 1.1 : Un message simple apparaît à l'écran

```
programme Hello;

debut
{création de la fenêtre}
{on écrit "HELLO" dans la fenêtre}
{destruction de la fenêtre}

{création de la fenêtre}
{on écrit "HELLO" dans la fenêtre}
{destruction de la fenêtre}

{création de la fenêtre}
{on écrit "HELLO" dans la fenêtre}
{destruction de la fenêtre}

fin.
```

Ce programme écrit "HELLO" à l'écran trois fois de suite. Au cas où vous n'auriez pas remarqué, le programme s'est un peu répété, en effet il comporte trois fois la même portion de code, ce qui fait trois fois plus de chances de se tromper en écrivant la même chose. Il doit sûrement y avoir une meilleure solution.

Procédures

Une *procédure* n'est rien d'autre qu'un groupement logique de déclarations de programme au sein d'un unique bloc de programme. Ce bloc de code peut être ensuite activé au moyen d'un appel de procédure. Autrement dit : vous prenez ces trois lignes de code et les mettez dans un paquet, puis vous donnez un nom à ce paquet et chaque fois que vous souhaitez appeler ces lignes de code, il vous suffit d'appeler ce bloc de code en utilisant le nom que vous lui avez donné. C'est ce que montre le Listing 1.2 :

Listing 1.2 : Procédure simple

```
procedure DitBonjour;

debut
{création de la fenêtre}
{on écrit "HELLO" dans la fenêtre}
{destruction de la fenêtre}
fin;
```

Les lignes de code sont maintenant regroupées dans la `procédure DitBonjour` (notre paquet). Maintenant, lorsque vous souhaitez ouvrir une fenêtre, afficher "Hello" à l'écran, puis détruire la fenêtre, il vous suffit d'appeler `DitBonjour`. Regardez notre nouveau programme qui figure dans le Listing 1.3 :

Listing 1.3 : Programme final (utilisant une procédure)

```
programme Hello;

procedure DitBonjour;
debut
{création de la fenêtre}
{on écrit "HELLO" dans la fenêtre}
{destruction de la fenêtre}
fin;
debut
      DitBonjour;
      DitBonjour;
      DitBonjour
fin.
```

Comme vous pouvez le voir, vous pouvez créer la procédure `DitBonjour` une fois pour toutes et l'appeler trois fois. Les occasions de commettre des erreurs dans cette procédure sont moins nombreuses, puisque vous n'avez écrit le bloc de code qu'une seule fois. Vous pouvez également réutiliser ce code en copiant-collant la procédure dans une autre application. Ce concept est illustré dans la Figure 1.3

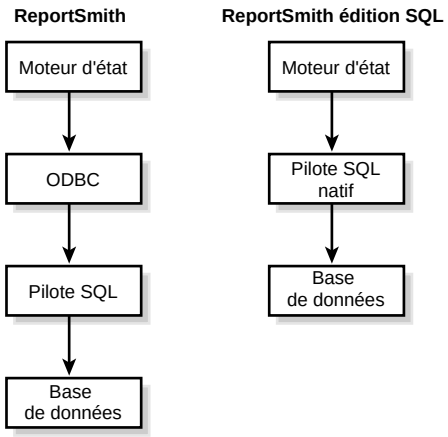


Figure 1.3
Réutiliser des procédures et du code.

Fonctions

Les *fonctions, c'est un peu différent*. Ce sont des procédures qui renvoient une valeur unique. Prenons un exemple. Soit l'équation y=cos(6). La fonction cos (cosinus) prend 6 comme opérande, calcule le cosinus de 6, puis renvoie cette valeur à l'équation. La valeur renvoyée est alors placée dans la variable y. Vous retrouvez là ce que vous avez pu rencontrer dans vos cours d'algèbre. Le concept de la résolution en y est fondamental. Vous pouvez remarquer que la fonction cosinus est appelée au sein même de l'expression mathématique. Nous touchons là le principe : l'appel de fonction lui-même devient la réponse une fois le retour de l'appel à la fonction. Ainsi, dans votre expression, les choses se déroulent comme suit :

Equation initiale : y=cos(6)

1. cos(6).
2. La fonction cos(6) calcule le cosinus de 6 et renvoie .99.
3. L'équation devient alors y=cos (avec cos=.99).
4. y=.99.

Les fonctions sont pratiques car vous ne pouvez pas appeler une procédure au milieu d'une expression. La raison en est qu'une procédure peut être définie de façon à renvoyer plusieurs valeurs à la suite de l'appel de procédure. Cette multiplication de valeurs dans une expression mathématique n'est pas faite pour arranger les choses. Les fonctions constituent donc une alternative élégante, puisqu'elles sont elles conçues pour ne renvoyer qu'une unique valeur.

Les unités : du code réutilisable

Les *unités* sont en fait un groupement de fonctions et de procédures qui s'appliquent à un même domaine. De même que vous pouvez regrouper des déclarations Pascal au sein d'une procédure ou d'une fonction, vous pouvez regrouper ces fonctions et procédures dans un niveau supérieur d'emballage, ce qui correspond à l'unité.

Examinons ensemble un exemple d'unité qui comporte trois fonctions : `DitBonjour`, `DitAurevoir` et `DitRien`. Le code de cette unité se trouve dans le Listing 1.4.

Listing 1.4 : Un exemple simple d'unité

```
unite DitKekChose;
debut
procedure DitBonjour;
debut
{création de la fenêtre}
{on écrit "HELLO" dans la fenêtre}
{destruction de la fenêtre}
fin;
procedure DitAurevoir;
debut
{création de la fenêtre}
{on écrit "BYE" dans la fenêtre}
{destruction de la fenêtre}
fin;
procedure DitRien;
debut
{création de la fenêtre}
{on écrit "RIEN" dans la fenêtre}
{destruction de la fenêtre}
fin
fin; {de l'unité DitKekChose}
```

La seule différence entre ce programme et celui du Listing 1.3 a consisté à regrouper trois procédures dans une unité appelée `DitKekChose`. Lorsque vous voudrez réutiliser ce code, il vous suffira de dire à votre programme `Use KekChose`. Si cette unité se trouve dans votre chemin d'accès (c'est-à-dire, si Delphi peut trouver l'unité `DitKekChose`), vous pourrez alors appeler ces trois procédures. Vous n'êtes pas obligé de réinventer la roue à chaque nouveau programme, comme c'était la règle autrefois dans le domaine de la programmation. En cela aussi, Delphi (et le Pascal objet) facilite la réutilisation du logiciel.

La Fiche

Au cœur de toute application Delphi pour Windows se trouve la fiche (*form* en anglais). Vous connaissez peut-être la fiche sous le nom de "fenêtre". La fenêtre est cet objet de base sur lequel est construit l'ensemble de l'environnement Windows. Si vous ne modifiez pas les paramètres par défaut, Delphi suppose que vous disposez d'une fiche dans chaque projet et affiche une fiche vierge chaque fois que vous lancez Delphi. Vous pouvez cependant choisir des options différentes dans Outils/Options. Vous pouvez enregistrer la position de vos fenêtres, ainsi que celles des fenêtres de code que vous avez ouvertes lors de la session précédente. Ces options sont propres à chacun de vos projets.

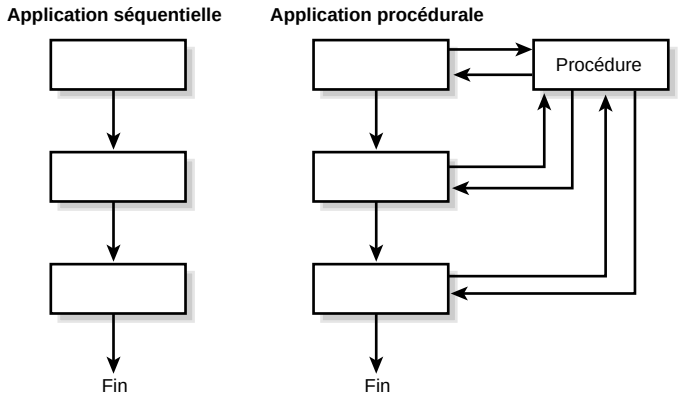
Vous avez peut-être déjà rencontré la fiche sous ses deux formes. Les fiches peuvent exister en état modal ou non modal. La fenêtre modale est celle qui se trouve au-dessus de toutes les autres fenêtres et qui doit être fermée pour accéder à une autre fenêtre. Une fenêtre non modale peut être mise de côté, et vous pouvez travailler dans d'autres fenêtres pendant ce temps. Nous traiterons tout au long de ce livre de la création des fiches sous toutes leurs formes. Ce sont les fondations de toute application Windows.

Propriétés de composant et de fiche

Les propriétés, dans l'acception Delphienne, sont les attributs d'un objet particulier, comme illustré Figure 1.4. Ainsi, un individu a pour attributs (entre autres) sa taille, son poids, la couleur de ses yeux et son numéro de Sécurité Sociale.

En Delphi, toutes les fiches et tous les Composants Visuels (ainsi que tous les contrôles ActiveX) ont des propriétés. Une fiche (ou une fenêtre) a une taille (hauteur et largeur), une couleur de fond, une bordure et d'autres attributs moins "visuels" tels qu'un nom. Nous pouvons contrôler l'aspect et le comportement de ces objets en modifiant ou manipulant leurs attributs ou propriétés.

Figure 1.4
Toute chose possède des propriétés.



Un exemple intéressant de composant visuel serait celui contrôlant un viseur de réalité virtuelle. Ce viseur ressemblerait à ceux que l'on peut trouver couramment. Les propriétés de ce viseur seraient des plus intéressantes. Un taux de rafraîchissement (comme celui d'un moniteur d'ordinateur), une résolution d'écran (éventuellement distincte pour chaque œil pour compenser d'éventuels défauts d'acuité visuelle), etc. Pendant l'éxécution de votre programme Doom VI, il vous serait possible d'ajuster constamment le viseur aux nouvelles conditions du jeu en modifiant les propriétés de l'objet viseur. Nous traiterons en détail de la question des propriétés dans cet ouvrage, et vous ne tarderez pas à apprécier toute leur utilité.

Application Delphi simple

Pour débuter votre carrière de développeur Delphi, nous allons voir dans cette partie comment construire votre première application Delphi. Nous allons supposer que vous avez déjà installé Delphi dans votre système Windows 95. Il vous suffit de suivre ces étapes :

1. Lancez Delphi en sélectionnant l'icône Delphi 3.0 dans le menu Démarrer/Programmes/Delphi 3.0.
2. Une fois Delphi lancé, vous voyez apparaître une fiche vierge intitulée FORM1. Si la fiche n'apparaît pas, sélectionnez Fichier/Nouvelle application dans la barre de menus principale de Delphi.
3. Choisissez l'onglet Standard dans la Palette des composants, à savoir la barre d'outils flottante située en haut à droite de l'écran.
4. Cliquez sur le contrôle Button.
5. Cliquez au milieu de la fiche. Un bouton devrait apparaître à cet endroit.
6. Double-cliquez sur le bouton. Une fenêtre de code devrait maintenant apparaître, comportant un curseur situé sur une ligne vide entre une ligne `Begin` et une ligne `End`.
7. Saisissez la ligne de code suivante dans la ligne vide :

```
Canvas.TextOut(20, 20, 'Avec Delphi 3, la programmation Windows
n'a jamais été aussi facile');
```

8. Vous venez de créer votre première application Delphi. Appuyez sur la touche F9 ou sélectionnez Exécuter/Exécuter dans le menu de Delphi pour compiler et exécuter votre application.

Vous devriez voir apparaître une fenêtre (il s'agit de la fiche omniprésente dont nous avons parlé plus haut). Le bouton que vous avez placé sur la fiche devrait se trouver là. Cliquez dessus. Vous pouvez remarquer que la phrase que vous avez entrée dans `Canvas.Textout` s'affiche dans la fenêtre. Voilà, la création d'une application Delphi n'est pas plus difficile que ça. Lorsque vous en avez fini avec votre nouvelle application, appuyez sur le bouton Fermer de la fenêtre (il s'agit du bouton comportant un X).

Vous pouvez maintenant sélectionner Fichier/Quitter, puis répondre Non lorsque Delphi vous demande si vous souhaitez enregistrer les modifications apportées à votre projet.

Aperçu de l'EDI de Delphi 3

Il est temps de s'attaquer à l'environnement de programmation de Delphi 3. Le but de cette section est de vous permettre de vous familiariser avec l'EDI de Delphi. Vous apprendrez également à personnaliser l'EDI en fonction de vos goûts.

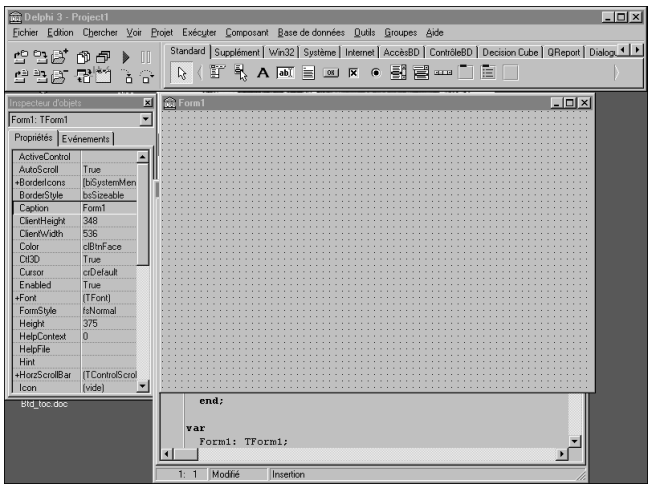
Nouveau

L'EDI, c'est-à-dire l'environnement de développement intégré (IDE en anglais), est un environnement dans lequel tous les outils nécessaires à la conception, à l'exécution et au test d'une application sont présents et fonctionnent harmonieusement afin de faciliter le développement de programmes. Dans Delphi 3, l'EDI consiste en un éditeur de code, un débogueur, une barre d'outils, un éditeur d'images et des outils de base de données, tous intégrés. Cette intégration met à la disposition du développeur un ensemble d'outils qui opèrent en harmonie et se complètent les uns les autres. Le résultat est un gain de temps et une diminution des erreurs dans le développement d'applications complexes.

Bases

Tout d'abord, il convient de lancer Delphi. Une fois Delphi chargé, votre bureau doit ressembler à la Figure 1.5.

Figure 1.5
L'EDI de Delphi.



Si vous ne voyez pas de fenêtre intitulée Form1 au milieu de votre écran, sélectionnez Fichier/Nouveau dans le menu principal de Delphi. L'EDI est composé de plusieurs parties principales, et la barre de menus de Delphi est elle-même constituée de deux parties principales.

Barre d'icônes

La *barre d'icônes* (voir Figure 1.6) permet un accès rapide et facile aux fonctions les plus utilisées. La configuration par défaut vous propose les 14 éléments que Borland considère comme les plus usuels. Ces éléments sont également accessibles à partir du menu de Delphi et ne figurent dans la barre d'icônes que pour en accélérer l'accès. Nous reviendrons en détail sur chacun de ces éléments une fois décrite la structure du menu de Delphi.

Figure 1.6
La barre d'icônes de Delphi.

Palette des composants

La palette des composants est le "catalogue visuel" de votre Bibliothèque de composants visuels (VCL). Elle vous permet de classer vos composants visuels dans des catégories appropriées. Par défaut, les composants sont regroupés en lignes fonctionnelles, c'est-à-dire, avec les composants d'accès aux données regroupés ensemble, et ainsi de suite. Ces regroupements ou *pages* sont distingués par des onglets. Les onze pages par défaut sont :

- Standard
- Supplément
- Win95
- AccèsBD
- ContrôleBD
- Win3.1
- Dialogues
- Système
- QReport
- ActiveX
- Exemples

Nous ne décrirons le bouton pointeur qu'une seule fois car on le retrouve sur chaque page de composants. Il est généralement enfoncé. Ce bouton vous permet de vous déplacer au sein des fiches et des fenêtres pour manipuler l'environnement Delphi. Lorsque vous sélectionnez un élément dans une des pages de composants, le bouton pointeur n'est plus enfoncé. Cela signifie que vous venez de passer dans un état où Delphi attend que vous placiez un composant sur une fiche. Une fois qu'un composant a été sélectionné, il suffit pour le placer sur une fiche de cliquer sur celle-ci.

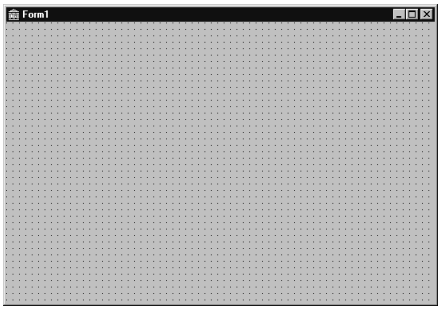
Pour placer le composant sur la fiche, vous pouvez également double-cliquer sur le composant, et celui-ci sera automatiquement placé sur la fiche. Si vous sélectionnez un composant puis changez d'avis, il vous suffit pour vous retrouver en mode Normal d'appuyer sur le bouton pointeur, qui est une sorte de bouton "Annuler" à utiliser pour le placement des composants. Le bouton pointeur se trouve toujours placé à l'extrémité gauche de la palette de composants, quelle que soit la page de composants sélectionnée.

Tous les composants seront décrits plus en détail lors du Jour 8, qui leur est consacré.

Fiche

La base de la quasi-totalité des applications Delphi est la fiche. Vous connaissez peut-être la fiche sous le nom de fenêtre, ce genre de fenêtre que vous pouvez rencontrer dans Word, Paradox ou d'autres applications Windows. En Delphi, la fiche est la fondation sur laquelle vous placez les autres composants Delphi. C'est la toile de fond de votre application Windows. Une fiche vierge Delphi type est représentée Figure 1.7.

Figure 1.7
Une fiche Delphi.



Une fiche Delphi possède les mêmes propriétés que toute autre fenêtre Windows 95. Elle comporte un menu de contrôle (situé dans le coin supérieur droit de la fiche), une barre de titre au sommet et, dans le coin supérieur gauche, les boutons Réduction, Agrandissement et Fermer. Vous pouvez masquer ces attributs de fiche si nécessaire, ou encore limiter leur utilisation.

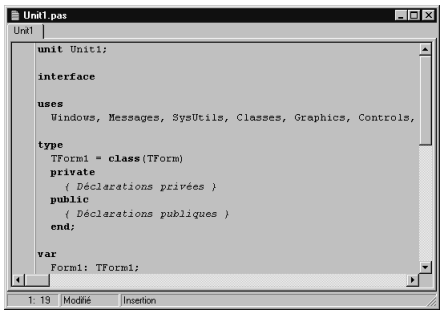
Note

Bien que pratiquement toutes les applications Delphi soient basées sur la fiche, vous pouvez utiliser Delphi pour écrire des DLL Win32 dans lesquelles les fiches peuvent ne pas exister du tout. Delphi peut également être utilisé pour écrire des composants visuels ne comportant aucune fiche.

Fenêtre d'édition

Un des éléments les plus importants de l'environnement Delphi est constitué par la fenêtre d'édition. Elle permet au développeur d'entrer le code Delphi. L'éditeur de code Delphi est un excellent éditeur aux fonctions très complètes (voir Figure 1.8). On y trouve notamment la mise en évidence colorée de la syntaxe (permettant de détecter plus rapidement d'éventuelles erreurs), des commandes d'édition de style Brief et la possibilité d'Annuler sans limitation.

Figure 1.8
La fenêtre d'édition.



Le nom du fichier actuellement en cours d'édition s'affiche dans la barre de titre de la fenêtre. Les onglets situés en haut de l'écran indiquent les pages utilisables. Les applications Delphi peuvent comprendre un grand nombre de fichiers source, et les onglets vous permettent de passer de l'un à l'autre.

Au bas de la fenêtre d'édition figurent trois éléments dignes d'intérêt. Le premier d'entre eux indique la ligne et la colonne du curseur dans le fichier en cours d'édition. Lorsque vous commencez un nouveau projet, le code que Delphi crée pour vous n'est pas sauvegardé. Vous devez l'enregistrer vous-même. Comme ce code a été modifié depuis la dernière sauvegarde (dans notre cas, il n'a jamais été enregistré), le mot `Modifié` apparaît à droite de l'indicateur de position. `Modifié` apparaît chaque fois que le code que vous voyez dans l'éditeur n'a pas encore été enregistré sur le disque dur. Dans notre cas, il n'y a pas de code enregistré sur disque, c'est pourquoi vous voyez apparaître cet indicateur. Le troisième élément indique si vous êtes en mode insertion ou refrappe. Cet indicateur est un grand classique de tous les outils d'édition.

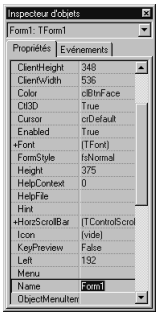
Inspecteur d'objets

L'inspecteur d'objets s'avérera indispensable pour votre travail en Delphi. Il propose une interface facile d'utilisation permettant de modifier les propriétés d'un élément Delphi, et de décider des événements auxquels réagira un objet.

L'onglet Propriétés

La partie Propriétés de l'Inspecteur d'objets (voir Figure 1.9) permet de consulter et de modifier les propriétés d'un objet. Cliquez sur la fenêtre de fiche vierge et regardez les attributs qui figurent dans l'onglet Propriétés de l'inspecteur d'objets. Petite remarque au passage : lorsqu'une propriété est précédée d'un signe plus, celle-ci comporte des sous-propriétés imbriquées.

Figure 1.9
L'onglet Propriétés de l'inspecteur d'objets.



Ainsi, vous pouvez remarquer qu'une fois la fiche sélectionnée, l'inspecteur d'objets comporte une propriété `Font` (Police) précédée d'un signe plus. Si vous double-cliquez sur cette propriété `Font`, elle s'ouvre pour révéler d'autres propriétés telles que `Color` (Couleur), `Height` (Hauteur), `Name` (Nom) et d'autres encore. Ce format offre un moyen clair, simple et pratique de modifier les attributs d'un objet.

L'onglet Evénements

L'onglet Evénements est l'autre facette de l'inspecteur d'objets (voir Figure 2.17). Il concerne le programmeur et les différents événements auxquels cet objet peut répondre. Ainsi, par exemple, si vous souhaitez que votre application fasse quelque chose de particulier au moment de la fermeture de la fenêtre, vous pouvez utiliser l'événement `OnClose` de la fiche à cet effet. Les événements et la programmation pilotée par événements seront traités en détail dans une prochaine section.

Figure 1.10
L'onglet Evénements de l'inspecteur d'objets.

Structure de menus de Delphi 3

La structure de menus de Delphi vous permet d'accéder à un ensemble complet d'outils. Jetez un œil à chacun des éléments de menu de Delphi et regardez leur fonction. Nous nous contenterons de passer rapidement en revue ces éléments et les parties suivantes rentreront plus dans les détails. Reportez-vous à la Figure 2.18 pour voir la barre de menus de base.

Info

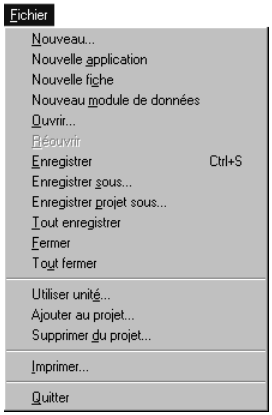
Les éléments de menu que vous pouvez voir dépendent en fait de votre version de Delphi. La version CS comporte certains ajouts que nous décrirons ici. Si vous souhaitez connaître précisément les différences entre les trois versions de Delphi 3, consultez votre manuel ou contactez Borland.

Figure 1.11
La barre de menus principale.

Fichier

Le menu Fichier est le menu contenant les éléments décrits ci-après (voir Figure 1.12).

Figure 1.12
Le menu Fichier.



Nouveau

En sélectionnant cet élément de menu, vous indiquez à Delphi que vous souhaitez créer un nouvel objet. Celui-ci peut être un membre quelconque du catalogue d'objets, comme un nouveau projet. La boîte de dialogue Nouveaux éléments apparaît lorsque vous sélectionnez

Nouveau, et vous permet de créer aussi facilement une nouvelle fiche qu'un nouveau serveur Web. Nous décrirons plus en détail certaines des options de cette boîte de dialogue par la suite.

Nouvelle application

En sélectionnant cet élément de menu, vous indiquez à Delphi que vous souhaitez créer un nouveau projet. Si aucun projet n'est actuellement ouvert, ou si le projet ouvert est à jour (s'il a été enregistré sur disque en son état actuel), Delphi referme le projet courant et en crée un entièrement nouveau. Il crée ainsi une nouvelle fenêtre d'édition de code (avec un nouveau fichier Unit1.PAS) et un nouvel objet Fiche (Form1), et fait apparaître l'inspecteur d'objets.

Nouvelle fiche

En sélectionnant cet élément de menu, vous indiquez à Delphi que vous souhaitez créer une nouvelle fiche. Par défaut, la fiche est une fiche vierge (et non pas une des fiches spéciales du catalogue d'objets).

Nouveau module de données

En sélectionnant cet élément de menu, vous indiquez à Delphi que vous souhaitez créer un nouveau *module de données*. Un module de données est une fiche non visuelle qui peut abriter tous vos contrôles de données. C'est un élément important car vous avez besoin d'une fiche dans laquelle toutes les fiches visuelles viendront prendre leurs données, et ce rôle est tenu par ce module de données.

Ouvrir

En sélectionnant cet élément de menu, vous indiquez à Delphi que vous souhaitez ouvrir un objet. Cet objet peut être un module de code ou un projet complet. Le premier répertoire dans lequel Delphi cherchera le projet est le répertoire de travail spécifié lors de l'installation de Delphi.

Réouvrir

Cet élément affiche à droite du menu une liste des derniers projets ou fichiers que vous avez ouverts. Son rôle est de vous faire gagner du temps.

Enregistrer

En sélectionnant cet élément de menu, vous demandez à Delphi d'enregistrer le module sur lequel vous êtes en train de travailler.

Enregistrer sous

Lorsque vous sélectionnez cet élément de menu, vous demandez à Delphi d'enregistrer le module courant sous un nouveau nom. Ceci est utile si vous avez l'intention de modifier radicalement une portion de code (si vous utilisez la version client/serveur de Delphi, vous pouvez

également utiliser le logiciel de contrôle de version PVCS). Vous pouvez ainsi conserver différentes révisions et revenir à un code plus ancien en cas d'erreur.

Enregistrer projet sous

En sélectionnant cet élément de menu, vous demandez à Delphi d'enregistrer le projet courant sous un nouveau nom. Vous pouvez ainsi enregistrer la totalité d 'un projet et le remiser pour un usage ultérieur.

Enregistrer tout

Cet élément de menu enregistre tout ce qui est ouvert : fichiers de projets et le reste.

Fermer

Cet élément de menu ferme le module de code ou la fiche associée actuellement sélectionnée. Si vous n'avez pas enregistré votre module dans son état actuel, Delphi vous demande si vous souhaitez enregistrer vos modifications.

Fermer tout

Cet élément de menu ferme le projet Delphi courant. Si vous n'avez pas enregistré votre projet dans son état actuel, Delphi vous demande si vous souhaitez enregistrer les modifications.

Utiliser unité

Cet élément de menu place une déclaration `uses` dans votre module de code courant, pour l'unité que vous souhaitez utiliser. C'est une manière pratique d'inclure des unités dans votre code sans être obligé de le faire manuellement.

Ajouter au projet

Sélectionnez cet élément de menu pour ajouter au projet Delphi une unité existante et sa fiche associée. Lorsque vous ajoutez une unité à un projet, Delphi ajoute automatiquement cette unité à la clause uses du fichier de projet.

Supprimer du projet

Sélectionnez cet élément de menu pour supprimer une unité existante et sa fiche associée du projet Delphi. Lorsque vous supprimez une unité d'un projet, Delphi supprime automatiquement cette unité de la clause uses du fichier de projet.

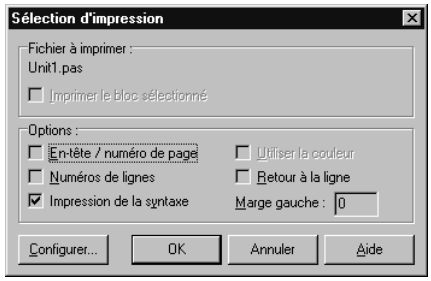
Imprimer

Sélectionnez cette option dans la boîte de dialogue de la Figure 1.13 pour imprimer l'élément actuellement sélectionné. Si cet élément est une fiche, Delphi vous propose diverses options d'impression. Cliquez sur OK pour lancer l'impression.

Figure 1.13
La boîte de dialogue Imprimer fiche.

Si l'élément que vous avez sélectionné est une fenêtre de code, plusieurs options vous seront alors proposées, parmi lesquelles l'impression des numéros de ligne, l'impression du texte sélectionné seul (si vous avez sélectionné du texte) et d'autres encore (voir Figure 1.14).

Figure 1.14
La boîte de dialogue Impression de sélection.



Quitter

Sélectionnez cet élément de menu pour sortir de l'EDI de Delphi. Si votre projet n'a pas été enregistré dans son état actuel, Delphi vous demande si vous souhaitez l'enregistrer avant de quitter.

Edition

Vous utiliserez les commandes du menu Edition pour manipuler le texte et les composants lors de la phase de conception. Il contient plusieurs éléments, comme indiqué Figure 1.15. Ces éléments sont passés en revue dans les sections qui suivent.

Figure 1.15
Le menu Edition.

Défaire/Récupérer

Cet élément de menu apparaît sous le nom Défaire ou Récupérer selon la dernière action effectuée. Si vous venez de supprimer un objet ou du code à l'aide de la touche Suppr, ou de l'option Supprimer dans le menu Edition, l'élément de menu s'appellera Récupérer. Si vous venez d'ajouter à votre projet du code ou des composants, l'élément de menu aura pour nom Défaire. Cette commande vous permet d'annuler vos derniers ajouts (idéal pour les remords de dernière minute).

Refaire

Refaire est l'inverse de Défaire. Refaire annule les Défaire que vous avez pu effectuer.

Couper

Sélectionnez cet élément de menu pour couper l'élément actuellement sélectionné (composant sur une fiche ou texte) et le placer dans le Presse-papiers. Les éléments sélectionnés sont enlevés de la fiche ou de l'unité de code courantes.

Copier

Sélectionnez cet élément de menu pour copier l'élément actuellement sélectionné (composant sur une fiche ou texte) dans le Presse-papiers. Les éléments sélectionnés *ne sont pas* enlevés de la fiche ou de l'unité de code courantes.

Coller

Sélectionnez cet élément de menu pour placer le contenu du Presse-papiers dans la fiche ou l'unité de code courante.

Supprimer

Sélectionnez cet élément de menu pour supprimer l'élément actuellement sélectionné. Cette suppression n'a rien de définitif, la commande Récupérer vous permet de corriger le tir en cas de suppression non désirée.

Sélectionner tout

Sélectionnez cet élément de menu pour sélectionner tous les composants de la fiche courante, ou tout le code de l'unité courante, selon celle des deux qui a le focus.

Aligner sur la grille

Cette option permet d'aligner sur la grille le composant sélectionné.

Info *Si l'option Aligner sur la grille est sélectionnée dans l'onglet Options/Environnement/Préférences, cet élément de menu n'est pas nécessaire. Tous les composants placés sur la page s'aligneront automatiquement avec la grille de la fiche.*

Mettre en avant-plan

Cela revient un peu à placer votre chouchou au premier rang de la classe. Lorsque vous placez de nombreux composants sur une fiche, il arrive qu'ils se recouvrent les uns les autres. Dans ce cas, il se peut qu'un composant soit recouvert par un autre alors qu'il devrait se trouver au-dessus. En sélectionnant ce composant puis cet élément de menu, vous placez le composant au-dessus de tous les autres.

Mettre en arrière-plan

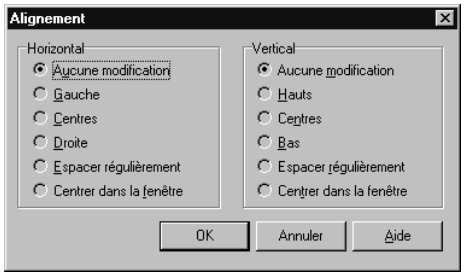
Ici l'effet est l'inverse de celui de l'élément Mettre en avant-plan. En sélectionnant cet élément, vous déplacez les composants sélectionnés derrière tous les autres composants.

Info *Les contrôles fenêtrés et non fenêtrés ne sont pas traités de la même façon. Tous les contrôles non fenêtrés sont placés "derrière" les contrôles fenêtrés.*

Aligner

Sélectionnez cet élément de menu pour faire apparaître la boîte de dialogue Alignement (voir Figure 1.16). Il vous est alors proposé différentes manières d'aligner vos composants verticalement et horizontalement sur votre fiche. Vous devez sélectionnez les éléments à aligner avant de sélectionner cet élément de menu.

Figure 1.16
La boîte de dialogue Alignement.



Taille

Sélectionnez cet élément de menu pour spécifier la hauteur et la largeur d'un composant (voir Figure 1.17). Si vous sélectionnez plusieurs composants, cet élément de menu vous permet de donner à tous les composants les dimensions (horizontales et/ou verticales) du plus grand composant sélectionné, ou du plus petit.

Figure 1.17
La boîte de dialogue Taille.

Echelle

Il arrive que l'on crée une fiche pour se rendre compte trop tard que tous les composants qu'elle contient sont trop grands ou trop petits. Grâce à l'option Echelle, vous pouvez mettre à l'échelle tous les composants de la fiche. En choisissant un nombre supérieur à 100 vous augmentez la taille, et vous la diminuez en choisissant un nombre inférieur à 100.

Ordre de tabulation

Delphi vous permet d'ajuster l'ordre de tabulation des éléments de la fiche sélectionnée. Cet ordre est celui dans lequel les objets récupèrent le focus lorsque l'utilisateur appuie sur la touche Tab pour passer d'un élément à l'autre. Vous pouvez ainsi imposer l'ordre de parcours des éléments afin de guider l'utilisateur dans ses saisies.

Lorsque vous sélectionnez cette option, Delphi place les noms de tous les composants de la fiche dans une zone de liste pour que vous les classiez visuellement (voir Figure 1.18). Cette façon de faire est bien plus simple que de devoir définir manuellement une propriété pour chaque contrôle.

Figure 1.18
La boîte de dialogue Ordre de tabulation.

Ordre de création

Sélectionnez cet élément de menu pour définir l'ordre dans lequel les composants non visuels sont créés. Quel en est l'intérêt ? Certains de vos composants non visuels peuvent dépendre d'autres composants non visuels qui sont présents et initialisés. Si les composants ne sont pas créés dans l'ordre adéquat, vous risquez de mettre la charrue avant les bœufs.

Verrouiller contrôles

Une fois votre écran conçu et vos contrôles placés, vous souhaiterez sans doute ajuster les propriétés et les événements. Il est très facile alors de laisser traîner la souris là où elle ne devrait pas et de déplacer involontairement un contrôle placé au millimètre près. Si vous sélectionnez l'élément Verrouiller contrôles, tous les contrôles sont verrouillés sur la fiche. Vous pouvez cliquer dessus pour modifier les propriétés et les événements sans risquer de les déplacer.

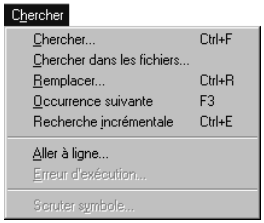
Ajouter à l'interface

Reportez-vous aux manuels ou à l'aide en ligne pour en savoir plus sur cette option de menu.

Chercher

Le menu Chercher est utilisé pour rechercher du texte, des erreurs, des objets, des unités, des variables et des symboles dans l'éditeur de code. Il contient plusieurs éléments, comme illustré Figure 1.19. Ces éléments sont décrits dans les sections suivantes.

Figure 1.19
Le menu Chercher.



Chercher

Delphi comprend un outil de recherche de grande qualité. Lorsque vous sélectionnez cette option, la boîte de dialogue qui apparaît vous propose de nombreuses options concernant la casse, la direction de recherche, etc. (voir Figure 1.20).

Figure 1.20
La boîte de dialogue Chercher texte.

Chercher dans les fichiers

Cette option vous permet de rechercher un texte spécifique et d'afficher chaque occurrence dans une fenêtre située au bas de l'éditeur de code. Les options de la boîte de dialogue vous permettent d'effectuer la recherche dans tous les fichiers ouverts, tous les fichiers du projet en cours, ou tous les fichiers à partir d'un répertoire particulier et dans les sous-répertoires.

Remplacer

La boîte de dialogue Remplacer complète la boîte de dialogue Recherche que nous venons d'évoquer. La différence entre les deux réside dans le fait que la boîte de dialogue Remplacer comporte une boîte d'édition Remplacer par, pour, justement, remplacer un morceau de texte par un autre.

Occurrence suivante

Cette option reprend la dernière recherche effectuée.

Recherche incrémentale

Il s'agit là d'une des plus séduisantes fonctions dont bénéficient les éditeurs Borland. Sélectionnez cet élément de menu, puis commencez à entrer un mot. A mesure que vous entrez les lettres, Delphi vous amène à la première occurrence de la chaîne de caractères tapée. Cet outil est très utile si vous savez à peu près ce que vous recherchez.

Aller à ligne

En sélectionnant cet élément de menu, puis en entrant un numéro de ligne (à hauteur du nombre de lignes de votre application), vous vous retrouverez à cette ligne.

Erreur d'exécution

A l'aide de cet élément de menu, vous pouvez entrer l'emplacement de votre dernière erreur d'exécution. Delphi compile alors votre application, puis s'arrête sur la ligne de code correspondant à cet emplacement. C'est également un moyen facile pour tracer les erreurs d'exécution. Cette fonction existe dans les divers Pascal Borland depuis des années.

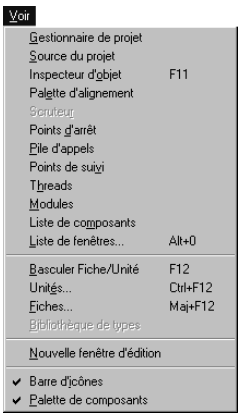
Scruter symbole

Une fois votre application compilée avec succès, vous pouvez en visualiser n'importe quel symbole. Ainsi, si votre application contient une fiche appelée Form1, vous pouvez entrer Form1 dans la boîte de dialogue Scruter symbole, et Delphi fera apparaître le scruteur de symboles contenant le symbole en question.

Voir

Les commandes du menu Voir permettent d'afficher ou de masquer différents éléments de l'environnement Delphi et d'ouvrir les fenêtres associées au débogueur intégré, comme indiqué sur la Figure 1.21. Ces éléments sont décrits dans les sections suivantes.

Figure 1.21
Le menu Voir.



Gestionnaire de projet

Sélectionnez cet élément de menu pour faire apparaître la fenêtre du Gestionnaire de projet. Ce Gestionnaire est traité en détail lors de la sixième journée.

Source du projet

En temps normal, vous ne voyez pas la routine Delphi principale qui lance l'application Delphi. Elle reste cachée car elle est la plupart du temps gérée et modifiée automatiquement. Vous pouvez afficher le code source pour ce morceau, mais il est préférable de ne pas le modifier sans savoir exactement ce que vous faites.

Inspecteur d'objets

Sélectionnez cet élément de menu pour faire apparaître l'Inspecteur d'objets, que vous avez vu Figures 1.9 et 1.10.

Palette d'alignement

Sélectionnez cet élément de menu pour faire apparaître la palette d'alignement. Cette palette est la version visuelle de la boîte de dialogue Alignement appelée par l'option de menu Edition/ Aligner. Il vous suffit de sélectionner les éléments à aligner, puis de faire apparaître cette palette et d'y sélectionner ce que vous souhaitez faire. Les conseils vous aideront si vous ne comprenez pas la signification des dessins.

Scruteur

Sélectionnez cet élément de menu pour faire apparaître une fenêtre permettant de visualiser le modèle d'héritages et les relations entre les objets. Ce scruteur est un outil puissant pour comprendre les véritables fondations objet de Delphi. Il vous permet de consulter la hiérarchie d'un modèle d'objet Delphi.

Points d'arrêt

Sélectionnez cet élément de menu pour faire apparaître une boîte de dialogue Liste des points d'arrêt. Celle-ci vous montre tous les points d'arrêt du débogueur que vous avez définis. Si vous cliquez à droite sur la boîte de dialogue, un menu apparaît qui vous permet d'ajouter, de modifier ou de supprimer des points d'arrêt.

Pile d'appels

Sélectionnez cet élément de menu pour faire apparaître la boîte de dialogue Pile d'appels. Cette boîte de dialogue vous indique l'ordre dans lequel les procédures et les fonctions sont appelées dans votre application. Cet outil sert lors du débogage.

Points de suivi

Vous pouvez visualiser et définir des points de suivi pour suivre des variables précises, ou pour créer des expressions utilisant ces variables. Lorsque vous définissez un point de suivi, vous pouvez également spécifier la façon dont le résultat du suivi sera affiché. Delphi affiche vos points de suivi dans les types appropriés (entiers affichés sous forme de nombre décimal, etc.).

Threads

Sélectionnez cet élément de menu pour afficher une liste des threads en cours d'exécution. Comme Windows 95 et NT sont tous deux des noyaux multitâches, vous pouvez lancer plusieurs threads à partir de votre application, afin d'effectuer plusieurs tâches indépendamment.

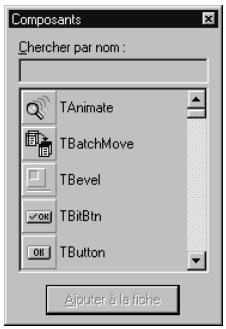
Modules

Cette commande affiche la boîte de dialogue Liste des modules, qui donne la liste de tous les modules utilisés dans le projet en cours.

Liste de composants

Sélectionnez cet élément de menu pour afficher la boîte de dialogue Liste de composants (voir Figure 1.22). Vous pouvez rechercher des composants par nom, ou faire simplement défiler la liste pour trouver celui qui vous intéresse. Si vous voyez un composant que vous souhaitez utiliser, appuyez sur le bouton Ajouter à la fiche, et le composant sera placé à l'écran.

Figure 1.22
La boîte de dialogue Liste des composants.



Liste de fenêtres

Il arrive qu'un grand nombre de fenêtres soient ouvertes simultanément et que vous ayez du mal à retrouver celle qui vous intéresse. En sélectionnant cette option, une boîte de dialogue apparaît qui présente une liste de toutes les fenêtres ouvertes par Delphi. Vous pouvez sélectionnez les fenêtres que vous souhaitez voir, et Delphi les placera à l'avant-plan.

Basculer Fiche/Unité

Lorsque vous travaillez sur une fiche, vous pouvez avoir à consulter le code associé à cette fiche, et vice versa. Cette option vous permet de basculer de l'une à l'autre.

Unités...

Sélectionnez cet élément de menu pour faire apparaître une boîte de dialogue montrant toutes les unités de votre projet. Vous pouvez alors cliquer sur l'unité que vous souhaitez voir, et l'éditeur de code l'affichera.

Fiches...

Sélectionnez cet élément de menu pour faire apparaître une boîte de dialogue montrant toutes les fiches de votre projet. Vous pouvez alors cliquer sur la fiche que vous souhaitez voir, et l'éditeur de code l'affichera.

Bibliothèque de types

Les Bibliothèques de type sont des fichiers composites OLE qui comprennent des informations relatives aux types des données, aux interfaces, aux méthodes et aux classes exportées par un contrôle ou un serveur ActiveX. Lorsqu'une bibliothèque de types est sélectionnée dans le panneau de liste des objets, il suffit de sélectionner bibliothèque de types pour voir apparaître une page d'attributs.

Cette page affiche des informations relatives à la bibliothèque sélectionnée. Les attributs et options suivants apparaîssent alors dans la page d'attributs : Nom, GUID, Version, LCID, Fichier d'aide, Chaîne d'aide, et Contexte d'aide.

Nouvelle fenêtre d'édition

Sélectionnez cet élément de menu pour ouvrir une nouvelle fenêtre d'édition, sans toucher à l'ancienne. L'unité courante située devant votre fenêtre d'édition est affichée dans la nouvelle fenêtre d'édition. Ceci vous permet de voir deux unités de code simultanément.

Barre d'icônes

Sélectionnez cet élément de menu pour rendre visible la barre d'icônes si elle ne l'est pas déjà.

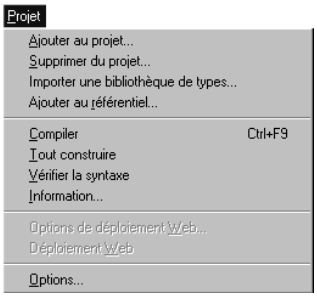
Palette des composants

Sélectionnez cet élément de menu pour rendre visible la palette des composants si elle ne l'est pas déjà.

Projet

Les commandes du menu Projet sont destinées à compiler ou à construire vos applications. Les commandes de ce menu ne sont disponibles que lorsqu'un projet est ouvert. Ce menu comporte plusieurs éléments, repris dans la Figure 1.23. Ces éléments sont décrits dans les sections suivantes.

Figure 1.23
Le menu Projet.



Ajouter au projet

Sélectionnez cet élément de menu pour ajouter au projet Delphi une unité existante et sa fiche associée. Lorsque vous ajoutez une unité à un projet, Delphi ajoute automatiquement cette unité à la clause uses du fichier de projet. Cette commande est identique à Fichier/Ajouter.

Supprimer du projet

Sélectionnez cet élément de menu pour supprimer une unité existante et sa fiche associée du projet Delphi. Lorsque vous supprimez une unité d'un projet, Delphi supprime automatiquement cette unité de la clause uses du fichier de projet. Cette commande est identique à Fichier/Supprimer du projet.

Ajouter au référentiel

Sélectionnez cet élément de menu pour ajouter la fiche courante au Référentiel d'objets. Ceci permet la réutilisation et vous fait gagner du temps en vous permettant d'utiliser et de réutiliser des fiches usuelles.

Compiler

Sélectionnez cet élément de menu pour compiler tous les fichiers qui ont été modifiés dans votre projet courant depuis le dernier exécutable produit.

Tout construire

Sélectionnez cet élément de menu pour reconstruire tous les composants, unités, fiches et tout le reste, que ces éléments aient changé ou non depuis le dernier exécutable produit.

Vérifier la syntaxe

Sélectionnez cet élément de menu pour vérifier que la syntaxe de votre application Delphi est correcte, sans être obligé de lier votre programme à un programme exécutable.

Information...

Sélectionnez cet élément de menu pour obtenir des informations concernant votre compilation Delphi et l'occupation de la mémoire.

Déploiement Web

Une fois la conception d'une ActiveForm (fiche active) terminée, sélectionnez cette commande pour l'envoyer sur votre serveur Web.

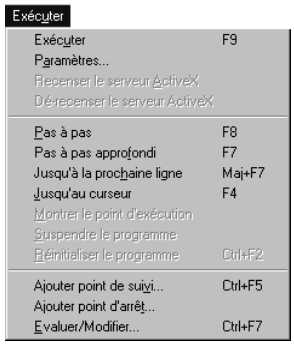
Options

Sélectionnez cet élément de menu pour faire apparaître la boîte de dialogue Options vous permettant de définir des options pour le compilateur, l'Editeur de lien (Lieur) et les répertoires.

Exécuter

Le menu Exécuter comporte des commandes qui vous permettront de déboguer vos programmes depuis l'EDI, comme illustré Figure 1.23. Ces commandes sont décrites dans les sections suivantes.

Figure 1.24
Le menu Exécuter.



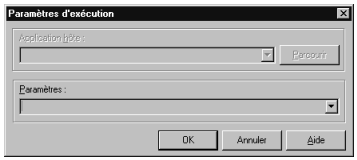
Exécuter

Sélectionnez cet élément de menu pour exécuter votre application Delphi. Si aucune compilation n'existe encore, Delphi commence par compiler l'application.

Paramètres...

Sélectionnez cet élément de menu pour fournir des paramètres en ligne de commande à votre application (voir Figure 1.25).

Figure 1.25
La boîte de dialogue Paramètres d'exécution.



Recenser le serveur ActiveX

Cette option de menu enregistre votre serveur ActiveX dans Windows 95. Il est alors possible d'utiliser votre contrôle ActiveX depuis un logiciel de navigation Web ou toute autre application. Avant de pouvoir utiliser pour la première fois un contrôle ActiveX, il faut l'enregistrer. Vous en apprendrez plus lors de la 13^{éme} leçon.

Dé-recenser le serveur ActiveX

Sélectionner cette option de menu permet de dé-recenser votre serveur ActiveX dans Windows 95 ou NT. De cette façon, vous pouvez supprimer de votre système l'instance de votre contrôle ActiveX. Vous en apprendrez plus à ce sujet au Jour 13.

Pas à pas

Sélectionnez cet élément de menu pour exécuter votre application ligne de code par ligne de code, tout en exécutant les appels de procédures et de fonctions comme s'ils n'étaient qu'une ligne de code. Ceci est pratique si vous souhaitez voir comment se comporte votre application, sans vouloir forcément voir le fonctionnement interne des appels de procédures et de fonctions.

Pas à pas approfondi

Sélectionnez cet élément de menu pour exécuter là aussi votre application ligne de code par ligne de code, mais en exécutant cette fois les appels de procédures et de fonctions ligne par ligne. Ceci est pratique si vous voulez voir ce qui se passe jusque dans les moindres détails.

Jusqu'à la prochaine ligne

Sélectionnez cet élément de menu pour voir quelle sera la ligne de code qui sera exécutée juste après. Cette fonction est complémentaire de Pas à pas approfondi.

Jusqu'au curseur

Sélectionnez cet élément de menu pour exécuter l'application jusqu'au point où se trouve le curseur dans votre code source. Arrivé là, vous pouvez utiliser la fenêtre Points de suivi pour évaluer une variable sur laquelle vous avez des doutes.

Montrer le point d'exécution

Sélectionnez cet élément de menu si, ayant fermé la fenêtre d'édition, vous vous trouvez au milieu d'un pas à pas en mode débogage dans votre application. Cette option de menu vous ramène dans une fenêtre d'édition où le curseur se trouve sur la ligne de code qui sera exécutée juste après.

Suspendre le programme

Sélectionnez cet élément de menu pour suspendre votre application et utiliser des points de suivi.

Réinitialiser le programme

Sélectionnez cet élément de menu pour arrêter un programme suspendu et le retirer de la mémoire.

Ajouter point de suivi

Sélectionnez cet élément de menu pour ajouter un point de suivi à la liste des points de suivi.

Ajouter point d'arrêt

Sélectionnez cet élément de menu pour ajouter un point d'arrêt à la liste des points d'arrêt. Cette option fait aussi apparaître un petit signe de stop rouge dans votre code source là où se trouve le point d'arrêt.

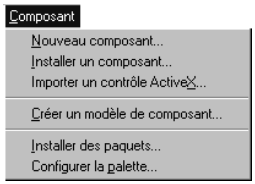
Evaluer/Modifier

Vous pouvez non seulement regarder les variables, mais également, grâce à cette option de menu, modifier en cours de route la valeur d'une variable. De plus, vous pouvez entrer une expression utilisant les variables de votre application, et cette expression sera immédiatement évaluée.

Composant

Le menu Composant permet d'ajouter et de configurer des composants ActiveX dans votre application Delphi. Comme vous pouvez le voir sur la Figure 1.26, Le menu Composant comporte plusieurs éléments, que nous passerons en revue dans les sections suivantes.

Figure 1.26
Le menu Composant.



Nouveau

Sélectionnez cet élément de menu pour faire apparaître l'Expert composant qui vous aide à créer un nouveau composant Delphi. Nous reviendrons sur la création de composants au Jours 13.

Installer

Ce composant vous permet d'ajouter de nouveaux composants visuels Delphi, ainsi que de nouveaux fichiers OCX à votre barre d'outils Delphi. Vous pouvez afficher une liste des composants actuellement installés et utiliser le bouton Ajouter pour en ajouter de nouveaux.

Importer un contrôle ActiveX

Sélectionner cette option vous permet d'importer un contrôle ActiveX (à condition qu'il soit déjà enregistré sur votre système) dans un paquet Delphi (nouveau ou existant).

Créer un modèle de composant

Cette option devient active lorsque vous selectionnez plus d'un composant sur une fiche. Par exemple, un composant `TTable` et un composant `TDatasource` peuvent être combinés en un seul qui peut être ajouté en une seule fois sur la fiche.

Installer des paquets

Cette option vous permet de déterminer quels paquets seront compilés à l'intérieur de votre application. Vous pouvez voir la liste des composants actuellement installés et ajouter, supprimer ou modifier le contenu de chaque paquet.

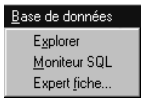
Configurer palette

Sélectionnez cet élément de menu pour ajouter ou supprimer des composants dans les onglets de la VCL. Ceci vous permet de mettre de côté les composants que vous n'avez pas l'intention d'utiliser dans votre projet.

Base de données

Le menu Base de données comporte plusieurs commandes vous permettant de créer, modifier et visualiser vos bases de données, comme indiqué Figure 1.27. Nous allons dans les sections suivantes passer en revue ces éléments.

Figure 1.27
Le menu Base de données.



Explorer

Sélectionnez cet élément de menu pour lancer l'Explorateur SQL qui vous permet de parcourir les structures de bases de données.

Moniteur SQL

Sélectionnez cet élément de menu pour lancer le Moniteur SQL. Ce moniteur vous permet de suivre les requêtes à mesure qu'elles sont exécutées dans votre application. Pour plus de détails, reportez-vous à l'aide en ligne ou aux manuels d'aide.

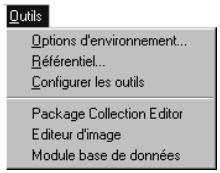
Expert fiche

Cet expert vous assiste lors de la création d'écrans d'entrée pour bases de données. Il commence par ouvrir la base de données à laquelle vous allez vous connecter, avant de vous aider à concevoir les écrans autour des données des fichiers. Pour plus de détails, reportez-vous aux chapitres consacrés aux bases de données.

Outils

Le menu Outils vous permet de voir et de modifier les paramétres d'environnement, de modifier la liste des programmes du menu Outils, et de modifier les modèles et experts. Il contient trois éléments (voir Figure 1.28), décrits dans les sections suivantes.

Figure 1.28
Le menu Outils.



Options d'environnement

Sélectionnez cet élément de menu pour faire apparaître la boîte de dialogue Options d'environnement. Cette boîte de dialogue vous permet de modifier les paramètres de l'éditeur, de l'affichage et de la palette, ainsi que les options du scruteur. Elle vous permet même de définir des options d'enregistrement automatique pour protéger votre travail.

Référentiel

Sélectionnez cet élément de menu pour visualiser les objets que vous avez placés à l'aide de la commande Projet/Ajouter au référentiel. Vous pouvez alors les supprimer ou en ajouter.

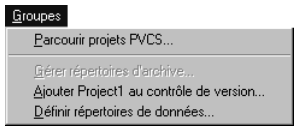
Configurer les Outils

Sélectionnez cet élément de menu pour personnaliser Delphi en ajoutant des outils à l'élément de menu Outils. Vous pouvez ainsi lancer votre application favorite à partir de là.

Groupes

Le menu Groupes contient les éléments de menu décrits dans la partie ci-après (voir Figure 1.29). Cet élément de menu n'existe que dans l'édition Développeur de Delphi (si vous avez acheté séparément PVCS) et dans la version Delphi C/S qui comprend PVCS d'origine.

Figure 1.29
Le menu Groupes.



Parcourir projet PVCS

Sélectionnez cet élément de menu pour faire apparaître la fenêtre de projet PVCS. Dans cette fenêtre, vous pouvez consulter les fichiers qui ont été au préalable enregistrés dans le système de contrôle de version.

Gérer répertoires d'archive

Sélectionnez cet élément de menu pour gérer la structure de répertoire de vos archives PVCS. Vous pouvez créer des répertoires afin de déplacer et d'organiser des fichiers.

Ajouter Project1 au contrôle de version

Sélectionnez cet élément de menu pour ajouter au système de contrôle des versions le projet sur lequel vous êtes en train de travailler. Vous en apprendrez plus sur la gestion de projets et l'utilisation de PVCS lors de la leçon du Jour 5.

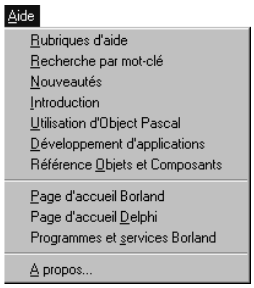
Définir répertoires de données

C'est là que vous pouvez définir vos répertoires de travail publics et privés.

Aide

Le menu Aide contient les éléments de menu décrits ci-après (voir Figure 1.30).

Figure 1.30
Le menu Aide.



Rubriques d'aide

Sélectionnez cet élément de menu pour accéder au fichier d'aide principal de Delphi 3, dans le format d'aide de Windows 95. Toutes les fonctions de recherche figurent dans le moteur d'aide, et il n'est donc pas besoin d'autres sélections de menu.

A propos...

Sélectionnez cet élément de menu pour voir le numéro de version du logiciel Delphi. Si vous voulez rire un peu, laissez cette fenêtre affichée, maintenez enfoncée la touche Alt, entrez `DEVELOPERS` et admirez le générique.

Personnalisation

Pour tailler Delphi à vos mesures, voici quelques modifications que vous pouvez apporter.

Barre d'icônes

Vous pouvez ajouter ou supprimer des éléments dans la barre d'icônes en cliquant avec le bouton droit sur cette barre, puis en sélectionnant Propriétés dans le menu contextuel. Ceci vous permet de personnaliser la barre d'outils.

Palette des composants

Vous pouvez également personnaliser la palette des composants en cliquant avec le bouton droit dessus, puis en sélectionnant Configurer dans le menu contextuel. Vous faites apparaître ainsi la même boîte de dialogue que vous auriez obtenue en appelant Options/Environnement/Palette. Elle vous permet d'ajouter, de renommer ou de supprimer un onglet, et de faire des ajouts à la palette des composants.

Fenêtre d'édition

La fenêtre d'édition peut elle aussi être personnalisée. Cliquez sur la fenêtre, puis choisissez Propriétés. Vous pouvez alors modifier la syntaxe, les mises en évidence colorées, et mêmes les touches utilisées par l'éditeur. L'éditeur prend en charge des commandes d'édition de style Brief (Brief est un éditeur vendu dans le commerce). Vous pouvez entre autres modifier le codage du clavier et utiliser des tabulations intelligentes.

Récapitulatif

Dans cette leçon, vous avez pu découvrir les premiers éléments de l'environnement Delphi. Vous avez vu ce que signifie RAD, vous avez découvert le genre d'outils que sont les VBX, les ActiveX et les VC, et de quelle façon on peut créer des VC avec Delphi. Une rapide introduction vous a présenté les constantes, les variables, les fonctions et les procédures. Nous avons vu ce qu'était un programmation pilotée par événements, en observant la façon dont Windows 95 réagit aux sollicitations. Enfin, nous avons parlé des propriétés et de ce qu'elles représentent dans Delphi. L'application toute simple que vous avez écrite s'étoffera au fil des leçons qui suivront.

Nous avons également passé en revue l'environnement de développement intégré (EDI) de Delphi. Vous avez peut-être été surpris par l'aspect très "liste des commissions" de cette leçon, mais cela semblait nécessaire. En effet, pour utiliser au mieux l'ensemble des outils proposés

par Delphi, il vous faut savoir quels sont les outils disponibles. Au fil de cet ouvrage, nous verrons plus en détail les divers outils et options, ainsi que les façons de les utiliser.

Une fois que vous aurez répondu au questionnaire et terminé les exercices, nous passerons à la leçon suivante consacrée au Pascal Objet.

Atelier

L'atelier vous donne trois moyens de vérifier que vous avez correctement assimilé le contenu de cette journée. La section Questions - Réponses reprend des questions posées couramment, et y répond, la section Questionnaire vous pose des questions, dont vous trouverez les réponses en consultant l'Annexe A, et les Exercices vous permettent de mettre en pratique ce que vous venez d'apprendre. Tentez dans la mesure du possible de vous appliquer à chacune des trois sections avant de passer à la suite.

Questions - Réponses

Q Comment gérer toutes les fenêtres qui apparaissent ?

R La meilleure chose à faire, si vous développez beaucoup, consiste à vous offrir un grand moniteur. Pour nous, le minimum acceptable est un moniteur de 17 pouces (en fait les trois auteurs de ce livre possèdent des moniteurs de 21 pouces).

Q Pourquoi a-t-on besoin de contrôles ActiveX et de contrôles visuels ?

R C'est en fait une question d'extensibilité. Delphi est parfait à l'heure actuelle, mais qu'est-ce qu'on fera une fois que les viseurs de réalité virtuelle se seront généralisés ? Il nous faudra un Composant visuel Delphi pour notre contrôle de viseur RV. En supportant l'ajout de nouvelles fonctions sous forme d'ajout à la barre d'outils, Delphi assure sa pérennité. Ce concept va même plus loin lorsqu'on en vient à parler de l'Internet. Microsoft (et d'autres éditeurs) proposent de nouveaux composants ActiveX chaque jour afin de vous permettre d'étendre vos compétences de développeur, et Borland vous aide à tirer parti de ces nouvelles ressources.

Q Il y a des produits qui ressemblent à Delphi Client/Serveur comme PowerBuilder. En quoi Delphi est-il différent ?

R Delphi Client/Serveur est radicalement différent, sur plusieurs points. Les dépliants publicitaires vous le diront mais il y a deux points importants : le Pascal objet comme langage de base place Delphi bien au-dessus de la mêlée, car il assure la plus solide des fondations. Par ailleurs, après observation, les applications PowerBuilder semblent bien plus lentes visuellement. Les exécutables Delphi sont plus rapides et — généralement — plus petits en taille que leurs équivalents PowerBuilder.

Questionnaire

1. Donnez deux avantages d'un environnement de développement RAD par rapport à un compilateur classique.
2. En quoi les contrôles visuels (VC) natifs de Delphi 3 diffèrent-ils des contrôles ActiveX ?

   En quel langage les contrôles visuels Delphi natifs sont -ils écrits ?

   Qu'est-ce que la mise en évidence de la syntaxe ?
5. Comment ajoute-t-on un nouveau composant visuel ou OCX à la barre d'outils ?

Exercices

1. Lisez la page d'introduction au Guide de référence du Pascal objet, décrivant un diagramme de syntaxe. Il est important que vous compreniez ce que sont les diagrammes de syntaxe. Si vous ne savez pas les lire, il vous sera difficile de rechercher la syntaxe d'une commande ou d'une expression. Ce manuel figure sur le CD-ROM de Delphi, et peut également dans certaines versions être fourni sur papier.
2. Créez une application comportant deux boutons et faites que le texte de `button1` soit recouvert par le texte de `button2` (le nouveau bouton).

2

Le Pascal Objet, première partie

Maintenant que vous avons vu comment l'EDI de Delphi peut vous faciliter la tâche, intéressons-nous au langage qui forme les fondations de Delphi : le Pascal objet. Le Pascal Objet est issu d'une grande lignée qui remonte à Turbo Pascal 1.0. A mesure de ses évolutions, le Turbo Pascal a vite été reconnu comme un des langage les plus riches (et un des compilateurs les plus rapides) du marché. Lorsque Borland présenta par la suite les extensions orientées objet au Turbo Pascal 5.5, ce fut la naissance d'une nouvelle génération de compilateurs. Ce que vous pouvez voir aujourd'hui dans Delphi 3 est l'accomplissement d'années de labeur consacrées à la création du langage parfait. Lors de cette journée, nous allons traiter des fondamentaux qui bâtissent le langage. Nous poursuivrons notre étude du Pascal Objet au Jour 3.

Attention *Vous devez saisir les listings qui figurent dans cette leçon et dans les suivantes en vous conformant à ces étapes :*

1. *Lancez Delphi.*
2. *Si un nouveau projet n'est pas automatiquement ouvert, choisissez Fichier/Nouveau projet dans le menu de Delphi 3.*
3. *Choisissez Projet/Options/Lieur et cochez la case Générer application console avant d'appuyer sur OK. Vous indiquez ainsi à Delphi que l'application que vous créez n'est pas une application fenêtrée mais une application de type "fenêtre DOS".*
4. *Sélectionnez Voir/Source du projet. La source maître du projet devient visible dans la fenêtre d'édition. Bien que vous n'éditiez pas ce code en temps normal, dans ce cas précis vous allez remplacer ce code par votre propre listing. C'est une façon simple de créer un programme.*
5. *Lorsque vous exécutez un programme, il apparaît dans une fenêtre qui lui est propre. Pour fermer cette fenêtre, appuyez sur Alt+F4 ou cliquez sur le X qui figure dans le coin supérieur droit de la fenêtre de sortie.*
6. *Attention : lorsque vous remplacez l'ancien code source par le vôtre, vous devez soit sauver auparavant le projet sous un nouveau nom (par exemple, `MaDemo`), soit ne pas modifier la ligne où figure `Program Project1`. En effet, Delphi gère lui-même la cohérence entre le nom de fichier et le nom du projet : si vous tentez de le faire à sa place, vous obtiendrez une erreur lors de la phase de compilation. L'astuce consiste donc à ne pas modifier le nom du programme, ou à enregistrer le projet sous le nouveau nom voulu (Fichier\Enregistrer projet sous).*

Un manuel de référence du Pascal Objet est fourni avec Delphi 3. Si vous ne l'avez pas sous forme papier, il est inclus dans l'aide en ligne. Il traite de tous les concepts abordés ici, et d'autres. Nous avons fait figurer dans ce livre les Jours 2 et 3 pour qu'ils tiennent lieu d'une sorte de "Reader's digest" du manuel de référence. Lorsqu'il s'agit d'apprendre le Pascal Objet, une deuxième source d'informations peut éclairer bien des points obscurs.

Le signe égal en Pascal

Avant d'aller plus loin, ceux d'entre vous qui connaissent d'autres langages de programmation, voire aucun, devraient se pencher sur l'utilisation du signe égal sous Delphi.

Ce signe peut revêtir deux significations. La première est l'opérateur d'affectation. Pour cette fonctionnalité, Delphi utilise := comme opérateur. Par exemple, `y := ax + b`. Il ne faut pas croire que cette instruction signifie "y possède la valeur de a fois x plus b" ; elle signifie "y prend la valeur a fois x plus b". Bien entendu, il ne doit pas y avoir d'incompatibilité de type entre ces différentes valeurs. N'oubliez pas que Delphi est un langage fortement typé, et qu'en conséquence vous devez être attentif au type des variables que vous manipulez.

La deuxième fonctionnalité en Delphi est la comparaison, ou encore la relation d'équivalence. Dans ce cas, on utilise directement le signe égal. Par exemple,

```
if A=B then WriteLn('A et B ont la même valeur')
```

Dans ce cas, vous n'affectez pas la valeur de B à A : vous testez l'égalité entre ces deux valeurs. Attention cependant au cas particulier de déclaration d'une constante :

```
Const
  TVA = 0.206;
```

Dans cet exemple, la constante `TVA` est équivalente à la valeur numérique `0.206`. Lors de la phase de compilation, le compilateur substituera à `TVA` la valeur `0.206` dans tout le code.

Constantes

Comme nous l'avons rapidement vu au Jour 1, lorsque vous commencez à programmer, vous vous apercevez vite que certains programmes nécessitent l'utilisation répétée de la même valeur. Un bon exemple d'un tel programme est un logiciel de facturation. Le taux de TVA risque d'y intervenir à maintes reprises. Pas de problèmes, il suffit de mettre 0.206 (20,6 %) cinq fois dans votre programme. Oui, mais les choses se gâtent lorsque le taux de TVA change. Vous devez alors vous souvenir de le changer aux cinq endroits où ce taux apparaît. Les risques d'erreur ou d'oubli sont alors multipliés d'autant.

C'est ici qu'interviennent les constantes. Les constantes ne sont rien d'autre qu'un nom donné pour une valeur que vous utilisez dans votre programme. Dans notre exemple, le nom `TVA` sera plus facile à utiliser que le nombre 0.206. Les constantes sont déclarées dans la partie `const` de votre application. Voici la nouvelle déclaration.

```
const
   TVA = 0.206;
```

Cette déclaration de constante a plusieurs utilités. Tout d'abord, elle rend le nombre plus facile à mémoriser. Si vous voyez 0.206 au milieu d'un listing, il n'est pas évident que vous compreniez tout de suite de quoi il s'agit. Ensuite, l'utilisation d'une constante facilite grandement le travail de maintenance d'une tierce personne. Il y a des chances que vous, le créateur de l'application, ne soyez pas forcément celui qui maintiendra le code. Si quelqu'un d'autre doit se pencher sur votre code pour modifier le taux de TVA, sa tâche sera facilitée s'il voit `TVA` au lieu de toute une série de 0.206 disséminés dans le code. Ainsi, les deux principaux bénéfices des constantes sont la lisibilité et la facilité de maintenance.

Astuce

Il est important que les noms de vos variables et constantes soient aussi descriptifs que possible. Les noms d'identificateurs peuvent être d'une longueur quelconque, mais seuls les 63 premiers caractères sont reconnus. Le premier caractère d'un nom d'identificateur doit être un caractère de soulignement ou une lettre. Les caractères qui suivent doivent être des lettres, des chiffres ou des caractères de soulignement.

Constantes en action

Les constantes peuvent prendre bien des formes et recouvrir de nombreux types de données. Voici un exemple utilisant différents types de données :

```
const
  MonNom = 'Ferdinand Bardamu';
  Age = 32;
  Salaire = 6.50;
  Adresse = '42, rue Ford Prefect';
```

La première constante, `MonNom`, est un exemple de constante de type `string` (chaîne de caractères). En Delphi 3, tout ce qui se trouve entre apostrophes, espaces y compris, fait partie de `MonNom`. Vous avez d'autre part défini `Age` comme étant égal à la valeur `integer` (entier) 32. La troisième constante, `Salaire` est égal au nombre `real` (réel) 6.50. Enfin, la quatrième constante est un autre exemple de valeur de type `string`.

Intéressons-nous à la vie d'une constante. Reprenons notre exemple de `TVA = 0.206`. Dans ce cas, lorsque vous demandez à Delphi2 de compiler votre application, Delphi regarde votre déclaration de constante, puis recherche toutes les occurrences de `TVA` dans votre application et les remplace par `0.206`. Toute l'astuce réside dans le fait que Delphi ne fait cela qu'au cours de la compilation et ne touche pas à votre code source. Les constantes sont là pour vous simplifier la vie, pas celle de Delphi. Delphi sait que vos déclarations de constantes ne sont en fait qu'un grand tableau de substitution.

Variables

Vous venez de voir l'utilité des constantes pour remplacer une donnée utilisée à maintes reprises dans votre application. Le problème avec les constantes est qu'elles ne peuvent changer de valeur au cours de l'exécution du programme. Les constantes doivent rester constantes. Leur utilité est limitée à des données qui ne changeront pas. Malheureusement, dans notre monde bien peu de choses restent constantes.

Heureusement, les variables sont là. Une variable ressemble à une constante en cela qu'il s'agit d'un nom donné à une valeur. Il y a une grosse différence cependant. La valeur d'une variable peut changer au cours de l'exécution du programme. La variable peut varier, d'où son nom. Une variable doit être définie sous un *type* donné.

Regardons un peu quels sont les différents types de données dans Delphi 3. Dans les paragraphes qui ont précédé, nous avons fait allusion à `real`, `integer` et `string`. Il s'agit là de quelques *types* de données parmi d'autres. Regardons-les plus en détail.

Types de données simples

La forme que prennent les données joue un rôle important dans la façon dont vous les percevez. Si un ami vous demande l'heure, vous ne lui répondez pas 2 345 567 secondes après minuit du 01/01/1970. Non, vous vous contentez de lui répondre "11h30".

En fait la forme que prend une donnée permet de la comprendre ou non. Pour ce qui est de la programmation, la façon dont les données sont entrées et enregistrées entrent également en considération. Comme vous ne disposez que d'une quantité de mémoire limitée sur un ordinateur, il est essentiel de ne pas en gaspiller. Par exemple, si vous souhaitez stocker le nombre 10, il serait dommage de le stocker dans un emplacement de mémoire assez grand pour contenir 1 999 999 999. Si vous faites ceci plusieurs fois (en attribuant un mauvais type et un mauvais emplacement mémoire) vous risquez d'occuper une grande portion de mémoire. Pour savoir quel emplacement vous allez allouer, il est également nécessaire de connaître l'intervalle des valeurs que la variable peut prendre.

Les types renvoient également à un autre concept. Le langage Pascal Objet est connu pour être un langage *fortement typé*. Cela signifie que Pascal s'assure que les données de types différents peuvent interagir de manière très structurée. Ce typage fort garantit que votre application n'essaiera pas d'ajouter votre âge à votre nom et de placer le résultat dans votre numéro de téléphone. Ce genre d'imbroglio est plus facile à faire dans des langages moins rigoureux. Certains se plaignent des contraintes imposées par les langages fortement typés, mais de tels langages vous assurent que votre application fera les choses comme il se doit. Ils limitent le programmeur dans ce qu'il doit faire, et pas dans ce qu'il peut faire.

Regardons quels sont les différents "types" de données et l'emploi qu'en fait Delphi 3 :

- Types entiers
- Types réels
- Le type `Currency`
- Types booléens
- Types caractère
- Types chaîne

Types entiers

Le type de données entiers est utilisé pour représenter des nombres entiers. Il existe plusieurs types capables de stocker une valeur entière. Le Tableau 2.1 montre ces types et leurs intervalles.

Tableau 2.1 : Les types de données Integer

Type	*Intervalle de valeur*	*Octets de mémoire nécessaires*	*Signé (peut contenir un nombre négatif)*
`Byte` (octet)	0...255	1	Non
`Word` (mot)	0...65535	2	Non
`ShortInt` (entier court)	–128...127	1	Oui
`SmallInt` (petit entier)	–32768...32767	2	Oui
`Integer` (entier)	–2147483648... 2147483647	4	Oui
`Cardinal`	0...2147483647	4	Non
`LongInt` (entier long)	–2147483648 à 2147483647	4	Oui

Vous pouvez remarquer que différents types entiers peuvent avoir des capacités de stockage radicalement différentes. Remarquez aussi que tout a un prix. La quantité de mémoire nécessaire augmente avec la capacité de stockage.

Info

L'espace mémoire est mesurée en octets. Un octet peut contenir 8 bits d'information. Un bit est un binaire `1` ou `0`. Les divers types de données de Delphi requièrent différents nombres d'octets pour stocker leur contenu. Le type `Byte` utilise un seul octet, le type `longint` en occupe quatre : chaque variable que vous définissez occupe de la mémoire, et cette mémoire est disponible en quantité limitée.

Integer est l'un des deux types génériques de Delphi 3. Les types *génériques* sont ceux qui sont affectés par l'unité centrale ou le système d'exploitation sur lesquels le compilateur est implémenté. Sous un système d'exploitation 32 bits tel que Windows 95, les types génériques ont leur capacité de stockage définie par le système d'exploitation.

Les entiers en action

Prenons un exemple simple d'utilisation d'un type `integer`. Le code du Listing 2.1 vous montre comment définir trois variables `Paie`, `PaieHeuresSupp` et `PaieTotale`.

Listing 2.1 : Programme de démonstration des types entiers

```
program ExempleEntiers;

uses
 Forms;

var
 Paie : Integer;
 PaieHeuresSupp : Integer;
 PaieTotale : Integer;

begin
 Paie := 5000;
 PaieHeuresSupp := 1000;
 PaieTotale := Paie + PaieHeuresSupp;
 WriteLn ('La paie totale est de FF', Paie Totale);
Readln {Pour que la fenêtre ne se ferme pas tant que vous n'avez pas appuyé
sur Entrée}
end.  {fin de ExempleEntiers}
```

Dans cet exemple, nous déclarons trois variables de type `integer`, nous affectons des valeurs numériques à `Paie` et `PaieHeuresSupp`, puis nous additionnons `Paie` et `PaieHeuresSupp` pour placer le résultat dans `PaieTotale`. Enfin, nous utilisons la procédure `Writeln()` pour placer le résultat à l'écran.

Type réels

Après le type entier, vient logiquement le type de données `real` (réel). Ces types de données real sont conçus pour contenir un nombre avec une partie fractionnelle. Dans l'exemple de code integer (Listing 2.1), nous avons supposé que le salaire serait en francs entiers. C'est rarement le cas. Dans le monde réel, on reçoit aussi des centimes, il est donc nécessaire d'utiliser un type de données qui reproduise cet aspect de la réalité. Nous revenons à ce que nous avons dit tout

à l'heure, concernant la nécessité d'utiliser le type de données adéquat. Vous pouvez choisir parmi différentes sortes de type real, comme le montre le Tableau 2.2.

Tableau 2.2 : Type de données réels

Type	*Intervalle*	*Octets de mémoire nécessaires*
`Real` (réel)	$\pm 2{,}9 \times 10^{-39}$ à $\pm 1{,}7 \times 10^{38}$	6
`Single` (simple)	$\pm 1{,}5 \times 10^{-45}$ à $3{,}4 \times 10^{38}$	4
`Double`	$\pm 5{,}0 \times 10^{-324}$ à $1{,}7 \times 10^{308}$	8
`Extended`	$\pm 3.4 \times 10^{-4932}$ à 1.1×10^{4392}	10
`Comp`	-2^{63} à 2^{63-1}	8

L'intervalle des valeurs que ces types peuvent accepter est impressionnant. Vous avez de quoi faire avant d'avoir besoin de générer un nombre qui dépasse $1{,}1 \times 10^{4392}$.

Info

Le type `Comp` est en fait un très grand nombre entier, et non un nombre réel. Ce type est inclus dans ce tableau parce qu'il est mis en œuvre de la même façon que les types à virgule flottante. Il s'agit en fait d'un entier à 64 bits.

Réels en action

Examinons un exemple mettant en œuvre des types de données réels. Nous allons reprendre notre programme de paie pour le rendre un peu plus réaliste. Regardez le Listing 2.2.

Listing 2.2 : Programme simple utilisant des types real

```
program ExempleReal;

uses
 Forms;

const
 Prelevements = 0.10;

var
 Paie : Single;
 PaieHeuresSupp : Single;
 PaieBrute : Single;
 PaieNette : Single;
```

```
begin
 Paie := 5000.55;
 PaieHeuresSupp := 1000.10;
 PaieBrute := Paie + PaieHeuresSupp;
 PaieNette := PaieBrute - (PaieBrute * Prelevements);
 WriteLn ('Le salaire brut est de ', PaieBrute,' FF');
 WriteLn ('Le salaire net est de ', PaieNette,' FF');
Readln {Pour que la fenêtre ne se ferme pas tant que vous n'avez pas appuyé
    sur Entrée}
end.  {de ExempleReal}
```

Cette application est un peu plus réaliste. Elle contient une variable `PaieBrute`, qui stocke justement la paie brute. La `PaieNette` est maintenant le résultat de `PaieBrute` moins le pourcentage de prélèvements défini dans Prélèvements. Nous avons également utilisé la constante `Prélèvements`.

Astuce

Essayez dans la mesure du possible d'utiliser des types de données `Single` ou `Double` plutôt que `Real`. Les `Real` sont plus lents à manipuler (il ne s'agit pas d'un type natif pour l'unité à virgule flottante du microprocesseur, chaque opération nécessite donc des conversions) et prennent plus de place ou sont moins précis que `Single` ou `Double`.

Type *Currency*

Un nouveau type de données qui mérite que l'on s'y intéresse est le type `Currency` (devise). Jusqu'à présent dans la plupart des langages, le développeur devait utiliser un type `Real` pour représenter des valeurs monétaires. Delphi 3 propose à cet usage spécifique un type `Currency`. Ce type est un type à virgule flottante qui est compatible dans ses affectations avec tous les autres types à virgule flottante, type `Variant` compris (nous reviendrons sur ce type par la suite). Le type `Currency` a une précision à quatre décimales et on le stocke sous forme d'entier à 64 bits (les quatre chiffres les moins significatifs représentent les quatre chiffres situés après la virgule).

Quel est l'intérêt d'un tel type ? Les principaux avantages sont au nombre de deux :

- Le type `Currency` est plus précis lorsque il s'agit de traiter de grands nombres.
- Le type `Currency` est utilisé dans `CurrencyField` et dans d'autres composants. Il est compatible avec les types de base de données représentant des devises.

Nous aurons l'occasion de réutiliser ce type au long de cet ouvrage.

Type *Currency* en action

Vous pouvez réécrire votre programme ExempleReal (Listing 2.2) en utilisant le type `Currency` pour les éléments monétaires. Le résultat devrait ressembler au Listing 2.3.

Listing 2.3 : Un exemple simple de programme utilisant Currency

```
program ExempleCurrency;

uses
 Forms;

const
 Prelevements = 0.10;

var
 Paie : Currency;
 PaieHeuresSupp : Currency;
 PaieBrute : Currency;
 PaieNette : Currency;

begin
 Paie := 5000.55;
 PaieHeuresSupp := 1000.10;
 PaieBrute := Paie + PaieHeuresSupp;
 PaieNette := PaieBrute - (PaieBrute * Prelevements);
 WriteLn ('Le salaire brut est de ', PaieBrute,' FF');
 WriteLn ('Le salaire net est de ', PaieNette,' FF');
Readln {Pour que la fenêtre ne se ferme pas tant que vous n'avez pas appuyé
sur Entrée}
end.  {de ExempleCurrency}
```

Types Booléens

Les type de données booléens sont les plus simples et les plus utilisés qui soient. Les variables de ce type représentent une quantité logique, `True` (vrai) et `False` (faux) par exemple. Vous pouvez alors vous demander pourquoi le Tableau 2.3 dresse la liste des cinq types booléens différents. Ceci s'explique par des raisons de compatibilité. Dans certains cas, Windows exige une valeur booléenne dont la taille est d'un word. Dans ces cas-là, d'autres types Boolean peuvent être utiles.

Tableau 2.3 : Les types de données Boolean

Type	*Intervalle*	*Octets de mémoire nécessaires*
Boolean	Booléen à un octet (le plus répandu)	1
ByteBool	Booléen à un octet	1
Bool	Booléen de taille word	2
WordBool	Booléen de taille word	2
LongBool	Booléen de taille deux words	4

A quoi sert un type Boolean ? Il s'applique à tout ce qui peut se décrire par OUI ou NON, VRAI ou FAUX, ARRET ou MARCHE.

Une des choses les plus importantes à remarquer est que les variables de type `Boolean` peuvent accepter les opérateurs `and`, `or` et `not`. Ceci vous permet de les manipuler plus librement.

Info

Si vous utilisez une `ByteBool`, `WordBool` ou un `LongBool` là où un type `Boolean` est nécessaire, Delphi 3 génère un code dans lequel toutes les valeurs non nulles de variables de ce type sont converties en 1 (`True`). Ceci vous permet d'utiliser les types de manière presque interchangeable.

Booléens en action

Les types booléens ont de nombreux usages, comme nous l'avons vu plus haut. Le Listing 2.4 est un programme d'exemple qui met en œuvre différents aspects des variables booléennes.

Listing 2.4 : Programme d'exemple simple sur les booléens

```
program ExempleBoolean;

uses
 Forms;

var
 FeuVert : Boolean;
 MonFlag  : Boolean;

begin
 FeuVert := False;
 MonFlag := not FeuVert;

 WriteLn ('Le flag FeuVert est défini comme ', FeuVert);
```

```
  WriteLn ('MonFLag est défini comme ', MonFlag);
  WriteLn ('Un ou logique sur ces deux flags donne ', FeuVert or MonFlag);
  WriteLn ('Un et logique sur ces deux flags donne ', FeuVert and MonFlag);
Readln {Pour que la fenêtre ne se referme pas tant que vous n'avez pas
    appuyé sur la touche Entree}
end.
```

Le Listing 2.4 montre comment créer deux variables, `FeuVert` et `MonFlag`, toutes deux de type booléen. On affecte ensuite une valeur à ces variables : `FeuVert` a pour valeur `False` et `MonFlag` a pour valeur l'opposé de `FeuVert`, soir `True`. On affiche à l'utilisateur la valeur des deux variables, puis on effectue un `or` et un `and` logiques (directement dans la déclaration `Writeln()`). Bien que cet exemple soit des plus simples, il illustre l'usage que l'on peut faire du type `Boolean` dans un vraie application. Nous utiliserons ce type à maintes reprises dans les leçons qui suivront.

Types caractère

Le type `Char` est sans doute bien connu de ceux d'entre vous qui ont programmé en C ou C++. Les types caractère ont été conçus pour ne stocker qu'un seul caractère. Un caractère a une taille d'un octet. Un rapide calcul vous montrera que 2^8(un octet) donne la possibilité de choisir parmi 256 caractères différents dans une variable de type `Char`. Si vous consultez la table ASCII qui figure à l'Annexe B de ce livre, vous verrez qu'il existe des caractères ASCII allant de 0 à 255 (en informatique, on commence généralement à partir de zéro, et non à partir de un).

Une des nouveautés apportées par Delphi 3 est l'ajout (ou plutôt la redéfinition des types caractère). Le type `Char` est maintenant l'équivalent du type `ANSIChar`. `ANSIChar` est toujours un caractère ANSI 8 bits. Par ailleurs, un troisième type de caractère, `WideChar`, vous permet d'utiliser un type de caractère 16 bits. A quoi bon trois types différents ? Cette fois encore, il s'agit d'une question de compatibilité. Delphi 3 prend en charge le standard `Unicode`, comme le montre le Tableau 2.4. Le type de données `WideChar` est le résultat de cette prise en charge. Un caractère `Unicode` utilise les 16 bits du type `WideChar`. Si vous placez une valeur normale `ANSIChar` dans une variable de type `WideChar`, l'octet de poids fort sera zéro et le caractère ANSI sera stocké dans l'octet de poids faible. Bien que Windows NT respecte parfaitement Unicode, ce n'est pas le cas de Windows 95. Si vous écrivez des applications destinées aux deux plates-formes, pensez à utiliser la fonction `SizeOf()` et ne partez pas du principe que les caractères ne font qu'un octet de long. Pour plus de détails sur Unicode, reportez-vous à l'Annexe D.

Info *A l'heure où cet ouvrage est mis sous presse, Borland a indiqué que le type de données `Char` est équivalent au type `ANSIChar`. Il a été suggéré que Char sera finalement équivalent au type `WideChar`. Si c'était le cas, tous les programmes utilisant le type de données Char respecteraient Unicode par défaut.*

Tableau 2.4 : Types de données character

Type de caractère	*Taille en octets*	*Contient*
`ANSIChar`	1	1 caractère ANSI
`WideChar`	2	1 caractère Unicode
`Char`	1	Pour l'instant égal à `ANSIChar`. Dans les versions futures de Delphi 3, ce type sera peut-être égal à `WideChar`

Info

Vous avez peut-être remarqué que dans Delphi 3 de nombreuses choses se plient aux définitions ANSI mais vont bientôt passer à leur équivalent Unicode. Les constructeurs s'éloignent peu à peu de Windows 3.1, et lorsque — conformément aux annonces de Microsoft — Windows 95 et Windows NT se fondront en un seul système, le monde NT 32 bits fera sa grande entrée. A ce moment-là, les développeurs de logiciels auront besoin d'un environnement de développement qui prenne totalement en charge Unicode. A cette date (et peut-être même avant) tous les types de données de Delphi 3 respecteront Unicode.

Types caractère en action

Pour un exemple de la façon dont fonctionnent les types de caractère, consultez le Listing 2.5.

Listing 2.5 : Un exemple de programme simple utilisant le type char

```
program ExempleChar;

uses
 Forms;

var
 Reponse : Char;
 Question : Char;

begin
 Question := 'A';
 Reponse := 'B';
 WriteLn ('La question est ', Question);
 WriteLn ('La réponse est ', Réponse);

 Reponse := Question;
 Question := #66;

```

```
  WriteLn ('La question est maintenant ', Question);
  WriteLn ('La réponse est maintenant ', Reponse);

  Readln {Pour que la fenêtre ne se ferme pas tant que vous n'avez pas
  appuyé sur Entree}

end.
```

Ce listing soulève plusieurs points intéressants. Après avoir déclaré deux variables de type `Char`, à savoir `Réponse` et `Question`, nous affectons une valeur à chacune d'elles. Votre déclaration d'affectation a affecté le caractère littéral "A" à `Question` et le caractère littéral "B" à `Réponse`. Une fois ces valeurs inscrites à l'écran vous faites un petit tour de passe-passe. Vous affectez à `Réponse` la valeur de `Question`. Vous pouvez procéder ainsi car les deux variables sont du même type.

La ligne qui peut sembler bizarre est celle-ci :

```
Question :=#66;
```

Souvenez-vous qu'une variable de type char ne peut contenir qu'un seul caractère. Si vous regardez quel est le caractère ASCII numéro 66, vous verrez qu'il s'agit de la lettre "B". Vous pouvez utiliser le signe # pour indiquer à Delphi 3 que vous souhaitez utiliser la représentation décimale d'un caractère plutôt que le caractère lui-même. Le type de données caractère est très utile et nous amène à passer au type suivant, le type de données `String` (chaîne de caractères).

Types chaîne

Le type de données `String` est plus souvent utilisé que le type `Char`. Dans Delphi 1, le type `String` était une concaténation de 255 caractères au maximum. En d'autres termes, il s'agissait d'un tableau de caractères. Delphi 3 gère les chaînes différemment. Le Tableau 2.5 rappelle les quatre types chaîne disponibles sous Delphi 3.

Tableau 2.5 : Les types chaîne

Type de chaîne	*Longueur*	*Contient*	*Terminaison nulle*
`ShortString`	255	`ANSIChar`	Non
`AnsiString`	jusqu'à env. 3 Go	`ANSIChar`	Oui
`String`	255 ou jusqu'à env. 3 Go	`ANSIChar`	Oui ou Non
`WideString`	jusqu'à env. 1,5 Go	`WideChar`	Oui

Delphi prend en charge les chaînes longues. Cette prise en charge est activée par la directive de compilation `$H+`. Cette directive est présente par défaut. Lorsque cette directive est utilisée, une variable du type `String` peut contenir une chaîne de longueur quasi illimitée (environ 3 Go).

Là aussi, Borland a donné au développeur le choix entre rester compatible avec Delphi 1.0 et suivre le progrès. Le type `String` est par défaut (avec la directive `$H+` activée) égale au type `AnsiString`. Le type `AnsiString` est une chaîne terminée par un `null` qui est allouée dynamiquement. Le grand avantage d'une variable de ce type est justement son allocation dynamique. A mesure que vous placez des chaînes plus longues dans cette variable, Delphi 3 réalloue la mémoire nécessaire à votre variable. Un autre avantage du type `AnsiString` est qu'il est déjà terminé par un `null`. Vous n'avez donc plus à utiliser les anciennes commandes de type `StrPCopy()` pour effectuer des conversions entre des chaînes de type Pascal (de longueur fixe) et des chaînes à terminaison nulle.

Quel est l'intérêt de ces terminaisons nulles ? Encore une question de compatibilité. Dans la plupart des appels aux routines systèmes, telles que l'API Win32, il est nécessaire de transmettre aux appels des chaînes à terminaison nulle. L'ancienne chaîne Pascal (désormais appelée `ShortString`) n'avait pas de terminaison nulle et devait être convertie avant d'être utilisée dans un appel API.

Delphi 3 assure la compatibilité avec Delphi 1.0 en proposant le type `ShortString`. Ce type est équivalent au type `String` de Delphi 1.0. Vous pouvez encore définir une chaîne de longueur spécifique, même si la directive `$H+` est invoquée. Voyez l'exemple qui suit :

```
{$H+} {Les chaînes longues sont maintenant activées}

var
  NouvelleString : String; {cette chaîne a une terminaison nulle et elle
                   est allouée dynamiquement }
  AncienneString   : String[20];  {En définissant la longueur de cette
  chaîne, Delphi 3 fait automatiquement de AncienneString un type
  ShortString, dont la longueur maximale est de 20 caractères}
```

Les composants VCL de Delphi 3 utilisent désormais le type `AnsiString` pour toutes les propriétés et les paramètres d'événements. Cela simplifie vos interactions avec les VCL et les API, en les uniformisant. Ainsi, vos applications interagissent parfaitement avec d'autres.

Types chaîne en action

Utilisons quelques chaînes pour nous familiariser avec elles. Reportez-vous au Listing 2.6.

Listing 2.6 : Exemple simple de programme utilisant des chaînes

```
program ExempleString;
```

```
uses
 Forms;

var
 Nom : String;
 Prenom : String[5];
 NouveauPrenom : String[30];

begin
 Nom := 'Roux-Combaluzier';
 Prenom := 'Marie';
 NouveauPrenom := 'Nathanaba';

 WriteLn ('Le prénom est ', Prenom);
 WriteLn ('Le nom est ', Nom);

 Prenom := NouveauPrenom;

 WriteLn ('Le nouveau prénom est ', Prenom);

Readln {Pour que la fenêtre ne se ferme pas tant que vous n'avez pas appuyé
sur Entree}
end. {ExempleString}
```

Vous avez défini une variable `Prénom` de type `String`. Comme vous n'avez pas spécifié de longueur, il s'agira d'une `AnsiString`. La variable `Prenom` que vous avez définie comme une `String` de longueur 5 (le 5 entre crochets [] indique la longueur de la chaîne), ce qui sera automatiquement converti en `ShortString[5]`. Cette chaîne contient désormais un maximum de cinq caractères. Enfin, `NouveauPrenom` a été défini comme une `String` de longueur 30, elle aussi automatiquement convertie en `ShortString`.

Mais il y a un problème avec ce code. L'avez-vous remarqué ? Regardez la ligne :

```
Prenom := NouveauPrenom;
```

Ce qui s'inscrit à l'écran devrait vous mettre la puce à l'oreille. Lorsque vous avez écrit `Prenom` à l'écran pour la deuxième fois, le prénom était Natha et pas Nathanaba. Pourquoi ?

La réponse se trouve dans la définition des variables. Vous avez défini `Prenom` comme une `String[5]`. Cette variable ne peut donc contenir que cinq caractères. Lorsque vous avez pris le contenu de `NouveauPrenom` (long de 8 caractères) pour le placer dans `Prenom` (qui ne peut contenir que 5 caractères), `Prenom` n'a gardé que les cinq premiers caractères. Les ordinateurs font ce que vous leur demandez, et pas forcément ce que vous voulez. La solution la plus évidente consiste à définir les deux variables comme des types `String` et à utiliser les possibilités

d'allocation dynamique du type `AnsiString`. Ainsi, vos variables contiendront tout ce que vous voudrez.

Dernière astuce concernant les types de données `String` : comment les vider ? Si vous souhaitez vider le contenu de la variable `NouveauPrenom`, il vous suffit d'écrire la ligne de code suivante :

```
NouveauPrenom := '';
```

En affectant à `NouveauPrenom` des apostrophes vides, vous définissez cette variable comme une chaîne vide. Simple, non ?

Structures de données

Jusqu'ici les types de données que nous avons vus servent à stocker une valeur unique, que ce soit un entier, un nombre réel ou un booléen. Nous allons maintenant passer à la vitesse supérieure. Les structures de données constituent un regroupement d'éléments de données voisins qui se trouve placé en mémoire. Ce regroupement d'éléments peut être traité élément par élément, bien que de nombreuses opérations puissent également être effectuées sur la structure de données dans son ensemble. Le Pascal Objet vous offre des déclarations de type permettant de créer vos propres structures de données à partir de types de données simples. Nous allons voir trois structures de données : les tableaux, les enregistrements et les ensembles. Dans les parties qui suivent, vous verrez comment les utiliser et à quel usage, et quels sont leurs avantages respectifs.

Info

Un type de donnée `String` appartient à deux catégories, selon le point de vue où vous vous placez. Si vous considérez les données de la chaîne comme une seule entité, il s'agit d'un type de données simple. Si vous considérez une chaîne comme un tableau de caractères (voir plus bas), il s'agit alors d'une structure de données.

Tableaux à une dimension

Les tableaux sont des outils. Ils vous permettent d'associer un nom de variable unique à toute une collection de données. Vous pouvez déplacer l'ensemble du tableau d'un endroit à un autre de la mémoire, le copier tout en référençant un seul nom de variable.

Si vous souhaitez manipuler un seul élément du tableau, vous identifiez cet élément en utilisant le nom du tableau et le rang de l'élément à traiter. Ainsi, `Noms[2]` identifie le deuxième élément du tableau `Noms`. Le grand avantage des tableaux est qu'ils permettent la réutilisation. Si vous placez une valeur dans une valeur unique, l'ancienne valeur est écrasée. Avec un tableau en revanche, vous pouvez stocker chaque nouvelle valeur dans un élément du tableau, pour un usage ultérieur. Ceci préserve les anciennes et les nouvelles valeurs. Chaque élément du tableau est stocké dans un emplacement de mémoire distinct et n'est pas affecté par son voisin.

Comme nous l'avons dit, vous identifiez l'élément du tableau à traiter en utilisant l'indice du tableau. Cet indice "repère" l'élément dans le tableau. Regardez la définition de tableau qui figure dans le Listing 2.7.

Listing 2.7 : Exemple de programme simple utilisant un tableau

```
program ExempleTableau;

Uses
 Forms;

type
 MonTypeTableau = array [1..5] of Real;

var
 ScoresPersonne : MonTypeTableau;

begin

 ScoresPersonne[1]:=55.6;
 ScoresPersonne[2]:=79.7;
 ScoresPersonne[3]:=42.2;
 ScoresPersonne[4]:=100.0;
 ScoresPersonne[5]:=34.9

WriteLn (ScoresPersonne[1]);
WriteLn (ScoresPersonne[2]);
WriteLn (ScoresPersonne[3]);
WriteLn (ScoresPersonne[4]);
WriteLn (ScoresPersonne[5]);

Readln {Pour que la fenêtre ne se ferme pas tant que vous n'avez pas appuyé
sur Entree}

end. {ExempleTableau}
```

Dans cet exemple, vous venez de créer un type de données appelé `MonTypeTableau`. Vous utilisez ce type de données tout comme vous utiliseriez `Integer`, `Real` ou un autre type simple. Le qualificateur `:4:2` est utilisé pour donner un format plus lisible aux données, qui sont de type `real`.

Une variable de type `MonTypeTableau` peut contenir cinq valeurs real. La partie `[1..5]` de la définition du type indique qu'il y aura cinq éléments, numérotés de un à cinq. Il est utile de le préciser car il est parfaitement possible de définir un tableau de cinq éléments avec pour indices `[6..10]`, aussi étrange que celui puisse paraître.

Vous chargez ensuite des données dans les éléments du tableau. Comme le tableau contient des valeurs real, vous affectez une valeur real à un élément spécifique dans le tableau (`tableau[élément]`). Vous pouvez remarquer que vous devez remplir chaque élément du tableau séparément. Ceci est nécessaire car chaque élément est véritablement indépendant, même si la structure de données dans son ensemble est référencée sous un nom unique (`ScoresPersonnes`).

Le Tableau 2.6 donne quelques exemples d'utilisation de tableaux.

Tableau 2.6 : Quelques utilisations de tableaux

Exemple de déclaration	*Résultat*
WriteLn ('Le premier score est ',Scores Personne[1]);	Affiche la valeur de l'élément 1 du tableau ScoresPersonnes, dont la valeur est de 55.6, dans une chaîne de texte qui apparaît à l'écran
Total := ScoresPersonne[2] + ScoresPersonne[5]/	Additionne les valeurs des éléments 2 et 5 (79. 7 et 42.2) et place le résultat dans une variable appelée Total
Total : = Total + ScoresPersonne[4]	
PersonScores[5] := ScoresPersonne[3] + ScoresPersonnes[1]	Prend les troisième et premier éléments du tableau ScoresPersonnes et place la somme des deux éléments dans l'élément numéro 5

Ce ne sont que quelques exemples d'utilisation des tableaux.

Tableaux multidimensionnels

Comme dans le monde réel, les tableaux sont souvent utilisés pour stocker une matrice de données. Regardez une feuille de calcul. Ce n'est rien d'autre qu'un tableau à deux dimensions. Les cellules peuvent contenir des données, comme c'est le cas dans un tableau.

Vous pouvez également définir des tableaux multidimensionnels. Vous pouvez voir de tels tableaux comme des cartes vous permettant de retrouver vos données. Si vous avez déjà joué à la bataille navale, vous savez ce que signifie "torpille en B7". Lorsque vous définissez votre type de tableau, vous devez indiquer à Delphi 3 à quoi ressemble la "matrice". Dans notre exemple, c'est une grille 3 par 3, comme celle d'un jeu de morpion. Le Listing 2.8 vous montre comment définir un tableau à deux dimensions.

Listing 2.8 : Exemple de programme utilisant un tableau à deux dimensions

```
program ExempleTableau2;

uses
 WinCrt;

type
 TypeSuperTableau = array [1..3, 1..3] of char;

var
 SuperTableau : TypeSuperTableau;

begin
SuperTableau[1,1] := 'X';
SuperTableau[2,2] := 'X';
SuperTableau[3,3] := 'X';
SuperTableau[3,1] := 'O';
SuperTableau[1,3] := 'O';

WriteLn (SuperTableau[1,1], ' ', SuperTableau[2,1], ' ',
  SuperTableau[3,1]);
WriteLn (SuperTableau[1,2], ' ', SuperTableau[2,2], ' ', SuperTableau[3,2]);
WriteLn (SuperTableau[1,3], ' ', SuperTableau[2,3], ' ', SuperTableau[3,3]);

Readln {Pour que la fenêtre ne se ferme pas tant que vous n'avez pas appuyé
  sur Entree}

end. {ExempleTableau2}
```

A en juger par ce que vous voyez à l'écran, qui a gagné au morpion ? Ce programme n'est pas très différent du précédent. Là aussi vous créez un type définissant l'aspect de votre tableau. La différence ici réside dans la définition du tableau elle-même. Vous devez maintenant définir les limites en deux dimensions. Pensez en termes de longueur × largeur, longueur × hauteur ou ligne × colonne, ou tout autre représentation qui vous aide à raisonner dans l'espace.

La définition `array[1..3, 1..3] of char` signifie :"Définissez un tableau dont les dimensions sont de trois cellules de large (étiquetées 1, 2 et 3) par trois cellules de long (étiquetées 1, 2 et 3) et faites que chacune de ces cellules puisse contenir une donnée de type `char`". Une fois le type défini, vous créez une variable, `SuperTableau`, de ce type. Vous pouvez à présent commencer à charger des valeurs dans le tableau. Puisque vous devez spécifier l'indice de chaque cellule que vous souhaitez remplir, vous devez jouer à la bataille navale... En disant `SuperTableau[1,1] := 'X'`, vous déclarez : "Prenez le caractère X et placez-le dans le

tableau `SuperTableau` à la ligne 1, colonne 1". Une fois que vous avez rempli un certain nombre de cellules du tableau avec des X et des O, la déclaration `WriteLn` affiche le plateau de morpion à l'écran.

Vous pouvez bien sûr pousser le principe beaucoup plus loin et définir des tableaux de dimension quelconque. Un tableau à six ou sept dimensions peut avoir son utilité. Une déclaration d'un tel type ressemblerait à ceci :

```
type
 TypeMegaTableau = array [1..5, 1..5, 1..5, 1..5, 1..5] of integer;

var
 MegaTableau : TypeMegaTableau;
```

Une fois cette définition faite, si vous souhaitez placer une valeur (dans ce cas une valeur entière), il vous suffit de donner les coordonnées des cellules à remplir, dans ces cinq dimensions. La ligne suivante conviendrait :

```
SuperTableau[3,2,4,2,1] := 2;
```

Cette ligne placerait la valeur entière 2 dans la cellule de la ligne 3, colonne 2, profondeur 4, continuum temporel 2 et mesure cosmique 1. Bon, j'avoue qu'on peut verser dans le bizarre dès que l'on essaye de donner des noms aux multiples dimensions d'un tableau, mais je pense que vous avez saisi le principe. Avant de vous lancer dans les espaces intersidéraux des tableaux à plusieurs dimensions, n'oubliez pas que la mémoire pour toutes les cellules, cellules vides y compris, est allouée au moment de l'exécution. Cela signifie que la mémoire de votre `MegaTableau` et les 3 125 cellules (c'est-à-dire 5^5 cellules) qui le composent seront allouées. Chacune de ces 3 125 cellules disposera d'un espace lui permettant de contenir une valeur entière. Si un entier fait 4 octets, cela fait 12 500 octets de mémoire, pour ce simple tableau. N'oubliez pas qu'il faut utiliser la mémoire avec discernement.

Enregistrements

Un autre type de structure de données très utile est le type de données enregistrement. Tout comme le tableau, l'enregistrement permet de stocker un ensemble d'informations se rapportant à un même sujet. Cependant, les éléments d'un enregistrement ne sont pas obligatoirement tous du même type. Les enregistrements sont parfaits pour stocker des informations concernant des gens ou des lieux. Vous pouvez stocker le nom, l'adresse, le numéro de téléphone et le poids d'une personne dans un enregistrement. Vous pouvez alors transmettre ce groupe d'informations en utilisant un nom unique, comme avec un tableau. Voyons ensemble un exemple de programme (voir Listing 2.9).

Listing 2.9 : Exemple de programme utilisant un type record

```
program ExempleEnregistrement;

uses
 Forms;

type
 TypeEnregPersonne = Record
       Nom : String[30];
       NumeroTel : String[13];
       Age : Integer;
       Sexe : Char
end; {TypeEnregPersonne}

var
 Personne : TypeEnregPersonne;

begin
 Personne.Nom := 'Dan Osier';
 Personne.NumeroTel := '(916)555-1212';
 Personne.Age := 32;
 Personne.Sexe := 'M';

 WriteLn ('Le nom de la personne est ', Personne.Nom);
 WriteLn ('Le numéro de téléphone de la personne est ', Personne.NumeroTel);
WriteLn ('L''âge de la personne est ', Personne.Age);
 WriteLn ('La personne est de sexe ', Personne.Sexe);

Readln {Pour que la fenêtre ne se ferme pas tant que vous n'avez pas appuyé
       sur Entree}

end. {ExempleEnregistrement}
```

Info *Vous pouvez remarquer que dans une des déclarations `Writeln` nous avons utilisé deux apostrophes de suite. Dans une chaîne littérale, deux apostrophes signifient qu'une apostrophe se trouve dans la chaîne (et qu'il ne s'agit pas d'une simple délimitation de la chaîne).*

Ce programme commence par définir le type. Ce type est une structure de données constituée de plusieurs parties. La première ligne, `TypeEnregPersonne = record` indique à Delphi 3 que les lignes de code qui suivront seront des éléments de la définition du type `record`. Il vous suffit alors de donner la liste des variables (et de leurs types) qui feront partie de l'enregistrement.

Ainsi `Nom` est une `String` de longueur 30, `NumeroTel` est une String de longueur 13, `Age` est un Integer et `Sexe` est un Char M ou F). La fin de la définition de votre enregistrement est marquée par une ligne `end;`.

Jusqu'ici, vous n'avez qu'un type `record`. Lorsque la variable `Personne` est déclarée (de type `TypeEnregPersonne`), vous avez effectivement alloué de la mémoire pour une instance de votre type `record`.

Maintenant que la mémoire est allouée, plaçons des données dans la structure de l'enregistrement. Vous ne pouvez pas dire `Personne := 'Dan';` car Delphi 3 ne sait pas dans quel élément de `Personne` il convient de placer la chaîne 'Dan'. Vous devez donc écrire `Personne.Nom := 'Dan Osier'`. Vous indiquez ainsi à Delphi 3 quel est le champ de l'enregistrement auquel vous souhaitez accéder. Cette *notation à point* (consistant à placer un point entre le nom de l'enregistrement et le sélecteur de champ) est un concept fondamental pour les enregistrements, mais plus encore pour la programmation orientée objet dont nous parlerons dans une prochaine leçon. Dans les lignes du programme qui suivent vous utilisez la notation à point pour sélectionner et remplir de données les différents champs de la variable `Personne`.

Une fois la variable `Personne` correctement remplie, vous pouvez en extraire les données, champ par champ. Vous pouvez remarquer que dans la procédure `WriteLn()` vous spécifiez quel est le champ que vous souhaitez afficher. `WriteLn` serait incapable de savoir quels champs imprimer si vous vous contentiez de dire `WriteLn (Person)`. Les enregistrements sont une structure des plus utiles. Pour faciliter l'écriture de votre application, il est important que vous utilisiez des types de données qui reflètent les données réelles. Dans cette optique, on peut considérer que l'enregistrement est une évolution logique du concept de tableau, permettant de stocker les attributs concernant une chose, une personne ou tout autre objet.

On complique un peu

Maintenant que nous avons vu les tableaux et les enregistrements, essayons de les combiner. N'ayez crainte, ça n'a rien de sorcier. Un tableau contient un grand nombre de choses de même type. Eh bien, votre `TypeEnregPersonne` est un type. Et si on définissait le tableau comme étant un `array[1..3] of TypeEnregPersonne` ? Oui, un tableau contenant trois enregistrements. Chacun de ces trois enregistrements peut contenir des informations concernant une personne, telles que son nom, son numéro de téléphone, etc. Le résultat est le programme qui figure dans le Listing 2.10.

Astuce

Lorsque vous entrerez le code du Listing 2.10, vous remarquerez que bon nombre de lignes sont très similaires. Au lieu de les taper toutes, il vous suffit de copier/coller dans la fenêtre d'édition et de remplacer le numéro de l'élément de tableau. Vous épargnerez ainsi vos nerfs.

Listing 2.10 : Exemple de programme utilisant des tableaux d'enregistrements

```
program ExempleTablEnreg;

uses
 Forms;

type
 TypeEnregPerson = record
       Nom : String[30];
       NumTel : String[13];
       Age : Integer;
       Sexe : Char
end; {TypeEnregPersonne}

 MonTypeTableau = array [1..3] of TypeEnregPersonne;

var
 tableauPersonne : MonTypeTableau;

begin

 tableauPersonne[1].Nom := 'Dan Osier';
 tableauPersonne[1].NumTel := '(916)555-1212';
 tableauPersonne[1].Age := 32;
 tableauPersonne[1].Sexe := 'M';

 tableauPersonne[2].Nom := 'Susie Smith';
 tableauPersonne[2].NumTel := '(916)555-9999';
 tableauPersonne[2].Age := 38;
 tableauPersonne[2].Sexe := 'F';
 tableauPersonne[3].Nom := 'Pat';
 tableauPersonne[3].NumTel := '(916)555-7766';
 tableauPersonne[3].Age := 30;
 tableauPersonne[3].Sexe := '?';

 WriteLn ('Le nom de la personne 1 est ', tableauPersonne[1].Nom);
 WriteLn ('Le numéro de téléphone de la personne 1 est ',
    TableauPersonne[1].NumTel);
 WriteLn ('L''âge de la personne 1 est ', tableauPersonne[1].Age);
 WriteLn ('Le sexe de la personne 1 ', tableauPersonne[1].Sexe);
```

```
WriteLn ('Le nom de la personne 2 est ', tableauPersonne[2].Nom);
 WriteLn ('Le numéro de téléphone de la personne 2 est ',
    TableauPersonne[2].NumTel);
 WriteLn ('L''âge de la personne 2 est ', tableauPersonne[2].Age);
 WriteLn ('Le sexe de la personne 2 est ', tableauPersonne[2].Sexe);

 WriteLn ('Le nom de la personne 3 ', tableauPersonne[3].Nom);
 WriteLn ('Le numéro de téléphone de la personne 3 est ',
    tableauPersonne[3].NumTel);
 WriteLn ('L''âge de la personne 3 est ', tableauPersonne[3].Age);
 WriteLn ('Le sexe de la personne 3 est ', tableauPersonne[3].Sexe);

Readln {Pour que la fenêtre ne se ferme pas tant que vous n'avez pas appuyé
sur Entree}

end. {ExempleTablEnreg}
```

Dans la partie de déclaration de types, vous commencez par placer la définition de l'enregistrement `TypeEnregPersonne`, puis utilisez ce type dans la définition de `MonTypeTableau`.

Attention *Vous devez définir quelque chose avant de l'utiliser. Si vous utilisez `TypeEnregPersonne` avant de le définir, le compilateur ne l'acceptera pas.*

Etudiez avec attention la définition du type `MonTypeTableau`. Maintenant, votre variable tableau `Personne` est un tableau d'enregistrements. Pour accéder à tableau`Personne`, vous devez lui donner l'indice du tableau (c'est-à-dire le nombre qui indique quel élément du tableau vous désirez), et l'identificateur de champ dans cet indice de tableau qui correspond à l'élément à visualiser ou à manipuler. Ainsi, la déclaration tableau `Personne[3].Nom := 'Pat';` dit en fait "dans la variable tableau `Personne`, prenez le troisième élément de tableau, puis dans cet élément le champ `Nom`, et rendez-le égal à la chaîne `'Pat'`.

On complique un peu plus

Les tableaux d'enregistrement sont bien, mais pourquoi s'arrêter en si bon chemin ? Pourquoi ne pas imaginer un enregistrement contenant d'autres enregistrements ? Ceci peut s'avérer très utile dans certaines applications.

Comme vous devez vous en souvenir, un enregistrement sert à regrouper un ensemble de données sur un même sujet, et dont le type n'est pas forcément le même. Un de ses éléments pourrait fort bien être un enregistrement. Prenons un exemple d'utilisation d'un tel enregistrement d'enregistrements. Un fichier des salariés contient des informations sur vous en tant que salarié. L'enregistrement contient sans doute les renseignements classiques, tels qu'on peut les voir dans les lignes suivantes :

```
type
 TypePersonne = record
   Nom : String[20];
   Prenom : String[10];
   NumeroSalarie : Integer;
 end; {TypePersonne}
```

Vous avez maintenant besoin d'ajouter l'adresse. Cette personne a sans doute une adresse de domicile et une adresse à son travail. La façon la plus simple de construire cet enregistrement consiste à créer un type d'enregistrement d'adresse universel, puis de le réutiliser. Un tel type figure dans les lignes ci-après :

```
type
 TypeAdresse = record
   Rue : String [50];
   Ville : String : [20];
   Departement : String [2];
   Code postal : String [10]
 end; {TypeAdresse}
```

Le grand avantage de ce type est que vous pouvez l'utiliser en plusieurs endroits. A présent, vos variables d'adresse de domicile et d'adresse de travail peuvent toutes deux être de type `TypeAdresse`. Regardons le code final (Listing 2.11) qui permet de me charger en tant que client :

Listing 2.11 : Autre exemple de programme utilisant des enregistrements

```
program ExempleEnregistrement2;

uses
 Forms;

type
 AddressType = record
   Rue : String [50];
   Ville : String : [20];
   Departement : String [2];
   CodePostal : String [10]
 end; {TypeAdresse}

 TypePersonne = record
   Nom : String[20];
   Prenom : String[10];
   NumeroSalarie : Integer;
```

```
    AdresseDomicile : AddressType;
    AdresseBureau : AddressType
  end;

var
  Salarie : TypePersonne;

begin

  Salarie.Nom := 'Osler';
  Salarie.Prenom := 'Don';
  Salarie.NumeroSalarie := 16253;

  Salarie.AdresseDomicile.Rue := '1313, rue Bakhounine';
  Salarie.AdresseDomicile.Ville := 'Kadath;
  Salarie.AdresseDomicile.Departement := '75';
  Salarie.AdresseDomicile.CodePostal := '75011';

  Salarie.AdresseBureau.Rue := '14, rue de Babylone.';
  Salarie.AdresseBureau.Ville := 'Interzone';
  Salarie.AdresseBureau.Departement := '75';
  Salarie.AdresseBureau.CodePostal := '75001';

  WriteLn(Salarie.Nom);
  WriteLn(Salarie.Prenom);
  WriteLn(Salarie.NumeroSalarie);

  WriteLn(Salarie.AdresseDomicile.Rue);
  WriteLn(Salarie.AdresseDomicile.Ville);
  WriteLn(Salarie.AdresseDomicile.Departement);
  WriteLn(Salarie.AdresseDomicile.CodePostal);

  WriteLn(Salarie.AdresseBureau.Rue);
  WriteLn(Salarie.AdresseBureau.Ville);
  WriteLn(Salarie.AdresseBureau.Departement);
  WriteLn(Salarie.AdresseBureau.CodePostal);

Readln {Pour que la fenêtre ne se ferme pas tant que vous n'avez pas appuyé
sur Entree}

end.
```

Dans votre enregistrement `TypePersonne` se trouvent deux variables, `AdresseDomicile` et `AdresseBureau`, qui sont toutes deux des variables d'enregistrement de `TypeAdresse`. Comme il s'agit d'enregistrements, vous devez spécifier la totalité du chemin d'accès qui permet d'accéder aux valeurs des variables. Ainsi, vous spécifiez la variable (`Salarie`), un point, le champ (`AdresseDomicile`), un point et comme `AdresseDomicile` est également un enregistrement, vous spécifiez également le champ dans `AdresseDomicile` (c'est-à-dire `Rue`). Le résultat est `Salarie.AdresseDomicile.Rue` et cette expression pointe sur une unique valeur chaîne. Cette méthode peut sembler fastidieuse mais elle est parfaitement lisible. Vous avez maintenant chargé la variable `Salarie` avec toutes les informations nécessaires, informations qui comprennent deux séries d'adresses. Pour afficher l'information à l'écran, vous devez récupérer chaque information séparément, puis utiliser `WriteLn()`.

Utiliser la notation à point revient un peu à donner votre adresse. Vous dites "Je vis en France, dans le département des Bouches du Rhône, dans la ville de Marseille, 8^{e} arrondissement, rue du Panier, numéro 17". En disant cela, vous rétrécissez peu à peu le champ jusqu'à ne pointer qu'à un seul endroit.

La seule façon de rendre tout ce processus un peu plus concis consiste à utiliser la clause `with`. Cette clause `with` définit en quelque sorte un paramètre par défaut, "Je vis en France" par exemple, ainsi toutes vos recherches commencent par la France au lieu du monde entier. La déclaration `Salarie.AdresseBureau.CodePostal := '75010'` n'est pas incorrecte, mais en utilisant la clause `with`, elle devient :

```
with Salarie do
 AdresseBureau.CodePostal := '75010';
```

Avez-vous remarqué que `Salarie` n'a pas été spécifié au début de la ligne `AdresseBureau.CodePostal ?` Ce renseignement était en fait impliqué par la déclaration `with`. Là où les choses se gâtent, c'est que le `with` ne fonctionne que sur la ligne de code qui le suit. Une seule ligne, me direz-vous, mais c'est sans intérêt ! Pas forcément : si cette ligne se trouve être une déclaration `begin`, tout ce qui se trouvera entre ce `begin` et le `end` qui lui est associé bénéficiera du `with`.

Nous allons améliorer notre programme en tirant partie de cette nouvelle découverte. Le code résultant figure dans le Listing 2.12.

Listing 2.12 : Encore un autre exemple de programme utilisant des enregistrements

```
program ExempleEnregistrement3;

uses
 Forms;

type
 TypeAdresse = record
   Rue : String[50];
```

```
    Ville : String[20];
    Departement : String[2];
    CodePostal : String[10]
  end; {TypeAdresse}

  TypePersonne = record
    Nom : String[20];
    Prenom : String[10];
    NumeroSalarie : Integer;
    AdresseDomicile : AddressType;
    AdresseBureau : AddressType
  end;

var
  Salarie : TypePersonne;

{Jusqu'ici le code est le même, c'est dans la suite qu'il change}

begin
with Salarie do
  begin {tout ce qui suit est associé au with}
    Nom := 'Osier';
    Prenom := 'Don';
    NumeroSalarie := 16253;
  end; {with Salarie}

with Salarie.AdresseDomicile do
  begin
    Rue := '1313 Your St.';
    Ville := 'MyTown';
    Departement := '75';
    CodePostal := '75011';
end; {with Salarie.AdresseDomicile}

with Salarie.AdresseBureau do
  begin
    Rue := '14 Big Business Road.';
    Ville := 'NoOzone';
    Departement := '75';
    CodePostal := '75011';
  end; {with Salarie.AdresseBureau}
```

```
with Salarie do
  begin
    WriteLn(Nom);
    WriteLn(Prenom);
    WriteLn(NumeroSalarie);
  end; {with Salarie}

with Salarie.AdresseDomicile do
 begin
   WriteLn(Rue);
   WriteLn(Ville);
   WriteLn(Departement);
   WriteLn(CodePostal);
 end; {with Salarie.AdresseDomicile}

with Salarie.AdresseBureau do
 begin
   WriteLn(Rue);
   WriteLn(Ville);
   WriteLn(Departement);
   WriteLn(CodePostal);
end; {with Salarie.AdresseBureau}
Readln {Pour que la fenêtre ne se ferme pas tant que vous n'avez pas appuyé
sur Entree}


end.
```

Vous pouvez remarquer à quel point le code est maintenant clair. Dans notre exemple, les déclarations `with` sont poussées d'un cran et s'appliquent à `Salarie.AdresseDomicile` au lieu du simple `Salarie`. Ceci permet d'aiguiller Delphi 3 directement sur ce qui vous intéresse et cela rend également le code plus facile à lire. Utilisez le code `with` à bon escient et vous améliorerez la lisibilité de votre code.

Intervalles

Lorsque vous pensez à un intervalle, vous le concevez peut-être dans un sens mathématique. Un intervalle de nombre peut aller de 1 à 10, de 30 à 30 000 et ainsi de suite. Un intervalle de lettres peut être "a" à "z", ou "A" à "F" (n'oubliez pas que dans Delphi 3 un "a" minuscule est différent d'un "A" majuscule car il s'agit de deux caractères ASCII distincts).

En Delphi 3, intervalles et sous-intervalles ont une signification proche de celle que l'on connaît dans le monde réel. Lorsque vous écrivez votre programme et que vous souhaitez compa-

rer ce qu'entre l'utilisateur pour voir s'il a utilisé des minuscules, vous employez des sous-intervalles.

Info

Un type de sous-intervalle est un intervalle de valeurs d'un type ordinal appelé le type hôte. Un sous-intervalle doit toujours définir la plus petite et la plus grande valeur du sous-intervalle.

Le type sous-intervalle peut facilement se construire et s'utiliser. Regardez le Listing 2.13 qui montre certains avantages des sous-intervalles.

Listing 2.13 : Exemple de programme utilisant un type sous-intervalle

```
program ExempleIntervalle;

uses
 Forms;

type
 LettreMinuscule = 'a'..'z';

var
 BonnesLettres : LittleLetter;

begin
 BonnesLettres := 'b';
 WriteLn (BonnesLettres)
Readln {Pour que la fenêtre ne se ferme pas tant que vous n'avez pas appuyé
sur Entree}

end.
```

Dans cet exemple, vous utilisez un sous-intervalle pour définir quelles valeurs sont acceptées pour affecter une variable de type `LettresMinuscules`. On crée alors une variable `BonnesLettres` de type `LettresMinuscules`. Cette variable ne peut contenir qu'un unique caractère compris entre a et z.

Vous vous demandez peut-être à quoi peuvent bien servir ces sous-intervalles. Pourquoi ne pas nous contenter de faire de `BonnesLettres` un type `char` ? Il y a une bonne raison : le Pascal Objet sait effectuer des vérifications d'intervalle. Cela signifie que lors de l'exécution d'un programme, lorsque vous faites une déclaration d'affectation telle que `BonnesLettres := 'b';` Pascal vérifie que la valeur placée dans `BonnesLettres` est bien conforme à son type (ou sous-intervalle). Si la valeur est hors de l'intervalle, une erreur d'intervalle (`range error`) survient. Ceci vous permet de détecter rapidement une erreur qui vous aurait échappé autrement. Les variables qui contiennent des choses comme les jours du mois (1..31) ou les mois de l'année (1..12) se prêtent

bien à l'utilisation de sous-intervalles. Ainsi, vous bénéficiez d'une vérification intégrée des valeurs transmises dans votre programme. Dès qu'une affectation illégale est tentée, le programme vous signale une erreur.

Pour bénéficier de cette vérification, il vous suffit de placer dans votre code un `{R+}` là où vous souhaitez que commence la vérification d'intervalle et un `{R-}` (facultatif) là où vous souhaitez qu'elle s'arrête. Vous pouvez également cocher la case correspondante dans la boîte de dialogue Options du projet, dans l'onglet Compilateur.

Info *La première règle qui régit la définition des sous-intervalles est que ces sous-intervalles doivent être d'un type ordinal, bien qu'un sous-intervalle d'entiers soit accepté. La seconde règle est que la valeur ordinale de la première entrée doit être inférieure à la valeur ordinale de la seconde entrée. Ainsi, un sous-intervalle de z à a n'est pas accepté, ce doit être de a à z car a a une valeur ordinale inférieure à celle de z.*

Ensembles

Les ensembles sont encore plus amusants à utiliser que les sous-intervalles. Les ensembles peuvent utiliser des sous-intervalles dans leur définition. Un ensemble est un groupe d'éléments que vous souhaitez associer sous un même nom et auquel vous comparez d'autres valeurs pour décider si ces valeurs doivent être incluses ou non dans l'ensemble. Par exemple, un ensemble peut contenir toutes les réponses (en un caractère) possibles à une question de type Oui/Non. Les quatre réponses possibles sont o, O, n et N. Vous pouvez créer un ensemble qui incluerait ces quatre réponses : ['o','O','n','N']. Une fois cet ensemble défini, vous pouvez l'utiliser pour voir si quelque chose appartient ou non à cet ensemble.

Vous déterminez si une valeur est incluse dans un ensemble à l'aide de la déclaration `in`. Ainsi, si vous créez la déclaration `Entree in ['o', 'O', 'n', 'N']`, vous dites que si la valeur de la variable `Entree` est un des éléments de l'ensemble `'o', 'O', 'n', 'N'`, cette déclaration sera alors évaluée comme étant le `Boolean True`. Si la valeur de `Entree` n'est pas dans l'ensemble, la déclaration est `False`. Un ensemble peut contenir pratiquement n'importe quoi, pour peu que les éléments de l'ensemble soit du même type ordinal ou de types ordinaux compatibles.

Les ensembles peuvent contenir des valeurs distinctes, mais aussi des intervalles. Ainsi, si vous écrivez un simulateur de téléphone, vous accepterez les chiffres de 0 à 9, et les touches * et #. L'ensemble correspondant sera donc [0..9, '*', '#']. Quand l'utilisateur appuie sur une touche, vous pouvez vérifier l'inclusion dans l'ensemble du caractère tapé en utilisant la déclaration `in`. Le Listing 2.14 vous montre un exemple d'utilisation d'ensembles.

Listing 2.14 : Exemple utilisant les ensembles

```
program DemoTel; Set Demo Program

uses
 Forms;
```

```
type
 TypeTouches = set of  Char;

var
 Touches : TypeTouches;
 EntreeUtilisateur : Char;

begin
 Touches := ['0'..'9', '*', '#'];
 Read(EntreeUtilisateur);
 If EntreeUtilisateur in Touches then
   WriteLn ('Cette touche est valide')
 Readln {Pour que la fenêtre ne se ferme pas tant que vous n'avez pas appuyé
sur Entree}

end.
```

Ce programme crée un type nommé `TypeTouches`. Remarquez que ce type est un ensemble de caractères. Il déclare ensuite une variable de type `TypeTouches`, appelé `Touches`. Cette variable peut contenir un ensemble de caractères. Dans la partie exécutable du code (celle qui est située après `begin`), on affecte à la variable `Touches` les caractères admissibles pour une saisie. Le test compare alors la saisie de l'utilisateur aux données contenues dans `Touches`. S'il existe une concordance, l'instruction `WriteLn()` est exécutée. Sinon, elle est ignorée. Le programme se termine par une instruction `ReadLn` afin d'empêcher la fenêtre de se refermer.

Constantes typées

Le concept de *constante typée* est étranger à nombre de développeurs. Une constante typée est en quelque sorte une variable préinitialisée. Vous pouvez définir une constante typée et lui donner une valeur dans la même déclaration, comme illustré dans cet exemple :

```
Const
 Max : Integer  = 88;
 Nom : String[10] = 'Dan';
 Chiffres : Set of '0'..'9';
```

Ceci peut sembler troublant de prime abord (*a priori* vous pensiez que la section de déclarations de constantes permettait de... déclarer des constantes). Il s'agit là d'une petite exception à la règle, une exception qui peut avoir ses avantages. Vous pouvez ainsi en une seule déclaration définir un type de variable et lui donner une valeur par défaut. La constante typée n'est initialisée qu'une seule fois avec la valeur par défaut, quel que soit le nombre de fois où le module contenant la déclaration est appelé.

Constantes typées en action

Les constantes typées sont faciles à utiliser. Le Listing 2.15 vous montre comment ces constantes typées sont utilisées tout comme des variables.

Listing 2.15 : Exemple de programme utilisant des constantes typées

```
program ExempleConstanteTypee;

uses
 Forms;

const
 MonNom : String = 'Dan Osier'

begin
 WriteLn ('Mon nom est ', MonNom);
 MyName:= 'Jim Fischer';
 WriteLn('Mon nouveau nom est ', MonNom);
 Readln

end.
```

Le programme commence en déclarant une constante typée appelée `MonNom`, de type `string`, et lui donne la valeur `'Dan Osier'`. Sa valeur est alors modifiée pour prouver qu'il est possible de modifier la valeur d'une constante typée au cours de l'exécution.

Types énumérés

Les types énumérés permettent d'améliorer grandement la lisibilité de votre code. Même si vous savez ce que fait votre code, une autre personne qui le lit ne le saura pas forcément. Imaginons que vous écriviez un programme qui gère les feux de circulation et que vous ayez besoin d'affecter à chaque couleur un nombre : 1 pour rouge, 2 pour orange et 3 pour vert. Dans votre application vous seriez tenté d'écrire if `3 then AllumePietonAttendez`. Ce n'est pas très parlant. Et imaginez un peu la tête du pauvre hère qui devra modifier votre application par la suite. Il regardera cette déclaration et se grattera la tête en se demandant ce qu'elle peut bien vouloir dire. C'est ici que les types énumérés interviennent.

Les types énumérés vous permettent de définir et d'utiliser un groupe d'objets appartenant à un ensemble. Il s'agit de ce que l'on appelle un type défini par l'utilisateur. Le nombre d'éléments composant un type énuméré est limité à 255.

Types énumérés en action

Le Listing 2.16 vous montre un exemple utilisant ces types énumérés.

Listing 2.16 : Exemple de programme utilisant des types énumérés

```
program ExempleEnumere;

uses
 Forms;

type
 CouleursFeu = (Rouge, Orange, Vert);

var
 Feu : CouleursFeu;

begin
 Feu := Rouge;
 WriteLn ('Le feu est ', Integer(Feu));
 Feu :=Vert;
 WriteLn('Le feu est passé au ', Integer(Feu));
 Readln

end.
```

Comme vous pouvez le voir dans le Listing 2.16, j'ai créé un type énuméré appelé `CouleursFeu`. Ce type a pour membres `Rouge`, `Orange` et `Vert`. On définit alors la variable Feu de type `CouleursFeu`. Par la suite, vous pouvez utiliser la variable `Feu` et lui affecter une des trois valeurs de couleur comme vous le faites dans la première ligne de code avec `Feu := Rouge;`. Les types énumérés ne sont pas là au bénéfice de l'utilisateur mais à celui du développeur qui obtient ainsi un code plus lisible et plus facilement modifiable. Bien que l'usage de ces types reste limité à l'intérieur du code, le travail de la personne chargée de modifier le programme sera facilité.

Type Variant

Le type `Variant` est l'un des ajouts les plus utiles dans Delphi. Ce type est très pratique car vous pouvez y placer n'importe quelle valeur, que ce soit un entier, une chaîne ou autre chose. Le type `Variant` est une structure de 16 octets qui contient la valeur et l'information de type associée. La souplesse de cette structure permet aux contrôles ActiveX de renvoyer une grande diversité de valeurs et de types de données, que le type `Variant` accepte toutes.

Delphi 1 ne comportait pas de type `Variant` du fait du caractère fortement typé du langage, mais aussi parce que le type `Variant` tend à favoriser des pratiques de programmation peu rigoureuses. Cependant, du fait de la demande et de nouvelles spécifications, Delphi 3 comprend maintenant un type `Variant`.

Type Variant en action

Le type `Variant` de Delphi 3 peut contenir une valeur entière, une chaîne ou une valeur à virgule flottante. Les usages de ce type sont aussi variés que le nom l'indique. Le Listing 2.17 donne un exemple d'utilisation.

Listing 2.17 : Exemple utilisant un type Variant

```
program ExempleVariant;

uses
 Forms;

var
  Entree : Variant;

begin
 Entree := 3.5555;
 WriteLn (Entree);

 Entree := 'Bonjour je m'appelle Raoul.';
 WriteLn (Entree);

 Entree := 4;
 WriteLn (Entree);
 Readln
end.
```

Ce programme utilise une variable appelée `Entree`, de type `Variant`. On lui affecte tout d'abord une valeur numérique, qui est ensuite affiché pour prouver que la variable peut contenir des valeurs réelles. On lui affecte ensuite une valeur de type chaîne, qu'on affiche ensuite, pour montrer que le type `Variant` sait également s'accomoder des chaînes de caractères. La démonstration se termine avec un entier.

L'atout majeur du type `Variant` est sa souplesse. Vous pouvez lui demander de stocker n'importe quel type de valeurs. Bien que Pascal Objet soit un langage fortement typé, vous retrouvez ainsi un peu de souplesse.

Opérateurs

Maintenant que nous avons vu les types de données, il est temps de voir comment comparer et évaluer des variables de ces types. Examinons ensemble les opérateurs pris en charge par Delphi 3 :

- Opérateurs arithmétiques
- Opérateurs logiques
- Opérateurs relationnels

Vous apprendrez également les règles de préséance qui gouvernent la vie des opérateurs quand ils sont plusieurs à composer une expression.

Opérateurs arithmétiques

Ces opérateurs vous permettent d'effectuer des opérations arithmétiques binaires et unaires. Un opérateur binaire utilise deux arguments, comme dans l'expression `A + B`, alors que les opérateurs unaires n'en utilisent qu'un, comme dans l'expression `not B`. Les Tableaux 2.7 et 2.8 décrivent les opérations, les types autorisés et le type résultant. Ainsi, si vous divisez un `Integer` par un Integer, le résultat est de type `Real`. On a parfois tendance à oublier ce genre de détails.

Tableau 2.7 : Opérateurs arithmétiques binaires

Opérateur	*Opération*	*Types utilisés*	*Type du résultat*
+	Addition	Integer	Integer
		Real	Real
-	Soustraction	Integer	Integer
		Real	Real
*	Multiplication	Integer	Integer
		Real	Real
/	Division	Integer	Real
		Real	Real
Div	Division entière	Integer	Integer
Mod	Reste	Integer	Integer

Tableau 2.8 : Opérateurs arithmétiques unaires

Opérateur	*Opération*	*Types utilisés*	*Type du résultat*
+	Identité du signe	Integer	Integer
		Real	Real
-	Négation du signe	Integer	Integer
		Real	Real

Opérateurs logiques

Les opérateurs logiques sont divisés en deux catégories : les opérations logiques et les opérations booléennes. Les opérations logiques consistent à décaler ou à comparer des choses au niveau du bit, tandis que les opérations booléennes consistent à comparer ou à manipuler des valeurs au niveau du `True` ou `False`. La liste de ces différentes opérations figurent dans les Tableaux 2.9 et 2.10.

Tableau 2.9 : Opérations logiques

Opérateur	*Opération*	*Types utilisés*	*Type du résultat*
`not`	Négation bit à bit	Integer	Boolean
`and`	ET bit à bit	Integer	Boolean
`or`	OU bit à bit	Integer	Boolean
`xor`	OU EXCL bit à bit	Integer	Boolean
`shl`	Opération	Integer	Boolean
`shr`	Opération	Integer	Boolean

Info

Si vous utilisez l'opérateur `not` sur un type Integer, le résultat sera dans le même type Integer. Si les deux opérandes d'un `and`, d'un `or` ou d'un `xor` sont de types `Integer`, le type du résultat sera celui commun aux deux opérandes.

Tableau 2.10 : Opérations booléennes

Opérateur	*Opération*	*Types utilisés*	*Type du résultat*
`not`	Négation	Boolean	Boolean
`and`	ET logique	Boolean	Boolean
`or`	OU logique	Boolean	Boolean
`xor`	OU EXCL logique	Boolean	Boolean

Opérateurs relationnels

Ces éléments, qui figurent au Tableau 2.11, permettent de comparer les valeurs de deux variables. Bien que nous n'ayons pas utilisé tous les types du Tableau 2.11, il a été jugé nécessaire de les y faire figurer pour information. Nous parlerons de certains des types de ce tableau dans la suite du livre.

Tableau 2.11 : Opérations relationnelles

Opérateur	*Opération*	*Types utilisés*	*Types du résultat*
=	Egal	Compatible simple, classe, référence de classe, pointeur, ensemble, chaîne ou chaîne compacte	Boolean
<>	Non égal à	Compatible simple, classe, référence de classe, pointeur, ensemble, chaîne ou chaîne compacte	Boolean
<	Inférieur à	Compatible simple, chaîne ou chaîne compacte, ou `PChar`	Boolean
>	Supérieur à	Compatible simple, chaîne ou chaîne compacte, ou `PChar`	Boolean
<=	Inférieur ou égal à	Compatible simple, chaîne ou chaîne compacte, ou `PChar`	Boolean
>=	Supérieur ou égal à	Compatible simple, chaîne ou chaîne compacte, ou `PChar`	Boolean
<=	Sous-ensemble de	Types d'ensembles compatibles	Boolean
>=	Sur-ensemble de	Types d'ensembles compatibles	Boolean

in	membre de	Opérande de gauche : n'importe quel Boolean type ordinal ; opérande de droite : ensemble dont la base est compatible avec l'opérande gauche

Si A et B sont des ensembles, leur comparaison donne les résultats suivants :

- A = B vaut True si A et B comportent exactement les mêmes éléments. Sinon, A <> B.
- A <= B vaut True si chaque élément de A est également élément de B.
- A >= B vaut True si chaque élément de B est également élément de A.

Préséance des opérateurs

Tout comme en arithmétique, vous devez savoir comment évaluer une expression. A cet effet, vous avez besoin de connaître l'ordre dans lequel les différentes parties d'une expression sont évaluées. La préséance des opérateurs affecte le résultat du code, tout comme il affecte le résultat d'un calcul en arithmétique (voir Tableau 2.12). Lorsqu'une expression comporte plusieurs opérateurs de même priorité, l'évaluation se fait de gauche à droite.

Tableau 2.12 : Précédence des opérateurs

Opérateurs	*Précédence*	*Catégories*
@, not	Premiers	Opérateurs unaires
*, /, div, mod, and, shl, shr, as	Deuxièmes	Opérateurs de multiplication
+, -, or, xor	Troisièmes	Opérateurs d'addition
=, <>, >, <, <=, >=, in, is	Quatrièmes	Opérateurs de relation

L'ordre est assez simple, mais vous devez garder en mémoire ces trois points :

- Un opérande entre deux séparateurs est lié (ou attaché) à l'opérateur dont la précédence est supérieure. Par exemple, dans l'expression `8^5-4`, `5` est attaché à l'opérateur `^`.
- Un opérande situé entre deux opérateurs égaux est lié à celui de gauche. Par exemple, dans l'expression `8-5-4`, `5` est attaché à l'opérateur `-` de gauche.
- Les expressions entre parenthèses sont évaluées avant d'être traitées en tant qu'opérande simple (on manipule d'abord le contenu des parenthèses). Par exemple, dans l'expression `8^(5-4)`, `5` est attaché à l'opérateur `-`.

Récapitulatif

Lors de cette journée, vous avez pu découvrir une partie du langage Pascal Objet. Vous avez vu les différents types de données dont vous disposez et, je l'espère, vous avez commencé à comprendre les mécanismes du langage. La meilleure façon de se familiariser avec le Pascal Objet est encore de l'utiliser. Regardez tous les exemples fournis avec Delphi 3. Lisez les sources, même si vous ne comprenez pas tout. Au Jour 3, nous poursuivrons notre présentation du Pascal Objet et traiterons de la structure d'un programme Delphi 3.

Atelier

L'atelier vous donne trois façons de vérifier que vous avez correctement assimilé le contenu de cette leçon. La section Questions - Réponses présente des questions qu'on se pose couramment et leurs réponses, la section Questionnaire vous pose des questions, dont vous trouverez les réponses en consultant l'Annexe A, et vous mettrez en pratique dans les Exercices ce que vous venez d'apprendre. Tentez dans la mesure du possible de vous appliquer à chacune des trois sections avant de passer à la suite.

Questions - Réponses

Q Est-il possible de convertir une variable d'un type à un autre ?

R Oui. Cela s'appelle le transtypage. A ce sujet, reportez-vous à l'exercice 1 du Jour 3.

Q Le Pascal Objet est-il identique aux Pascal qui existent sur d'autres plates-formes et qui sont proposés par d'autres éditeurs ?

R Il existe un Pascal ANSI standard. Borland a apporté beaucoup d'améliorations au Pascal standard, et au final, les avantages apportés rachètent largement la non conformité au standard.

Questionnaire

1. En quoi les constantes et les variables sont-elles différentes au niveau le plus fondamental ?
2. Pourquoi la précédence des opérateurs est-elle nécessaire ?
3. Quel est l'avantage apporté par les constantes typées ?

Exercices

1. Pour mieux vous familiariser avec le type `Variant`, écrivez une application qui utilise des types `Integer` et `Variant`. Assurez-vous que vous placez la valeur `Integer` dans la variable `Variant`. Essayez d'écrire les deux valeurs à l'écran pour voir si le type `Variant` contient vraiment d'autres types de données (Integer dans cet exemple).
2. Ecrivez un programme qui utilise les types énumérés. Essayez de créer un type appelé `MembreDeLaFamille` qui a pour éléments tous les membres de votre famille. Donnez l'âge de chacun de ces membres et affichez tout.

3

Le Pascal Objet, deuxième partie

Lorss de la leçon précédente, vous avez pu découvrir les types de données de Delphi 3 et leurs utilisations. Vous commencerez ici à voir comment contrôler vos applications. Le Pascal Objet fournit plusieurs commandes très puissantes vous permettant de contrôler le comportement de vos applications. Nous montrerons en quoi la modularisation de votre application permet de réduire le nombre d'erreurs et de gagner en fiabilité. Nous évoquerons aussi les pointeurs, ainsi que le passage de paramètre.

Contrôler le flux

Toute la programmation repose sur la capacité de votre application à prendre des décisions, en se basant sur des entrées ou d'autres critères, et sur sa capacité à effectuer ensuite une tâche ou une opération donnée. Cette capacité est généralement appelée l'exécution conditionnelle. Le Pascal Objet propose plusieurs commandes permettant de mettre en place cette "capacité conditionnelle". Examinons ensemble ces commandes.

If ... Then ... Else

La clause `If...Then...Else` est la déclaration conditionnelle la plus fondamentale. Elle permet au développeur de poser une question et d'effectuer une opération ou une tâche donnée selon la réponse obtenue. Comment demander à l'utilisateur d'entrer quelque chose et prévenir cet utilisateur si son entrée n'est pas correcte ? Commençons par un exemple utilisant du code séquentiel, puis nous le transformerons. Le Listing 3.1 montre comment poser une question à l'utilisateur sans structure conditionnelle.

Listing 3.1 : Programme simple où l'utilisateur entre des données

```
program ExempleIf;

Uses
 Forms;

var
 EntreeUtilisateur : Integer;

begin
 Write ('Quel âge avez-vous ?');
  ReadLn (EntreeUtilisateur);
  WriteLn (EntreeUtilisateur, ' est le plus bel âge !');
  ReadLn
end.
```

Cet exemple de programme est des plus simples. Examinons-le ligne par ligne. Vous déclarez une variable EntreeUtilisateur de type Integer. Elle permet de stocker l'âge de l'utilisateur, que celui-ci vous fournit. La ligne Write demande à l'utilisateur son âge. Vous utilisez la fonction Write() plutôt que WriteLn() car Write ne fait pas passer le curseur à la ligne suivante. La fonction ReadLn() accepte les données que l'utilisateur entre sur son clavier jusqu'à ce que la touche Entrée soit pressée. L'entrée est alors placée dans la variable EntreeUtilisateur, et l'exécution du programme se poursuit. Le WriteLn() final annonce à l'utilisateur que son âge est le plus bel âge.

Ce programme soulève quelques problèmes. Il n'effectue pas de vérification d'intervalle pour voir si l'âge entré se trouve dans les limites du raisonnable (il acceptera des âges négatifs ou supérieurs à 130). Utilisons donc la clause If...Then pour corriger ce petit défaut. La syntaxe de la clause est la suivante :

```
If <Expression1> then <Expression2> else <Expression3>;
```

La première expression doit avoir pour valeur un True ou False logique. La deuxième expression est l'action qui sera effectuée si la première expression est évaluée comme True (vraie). La troisième expression est facultative et elle indique l'action à effectuer si la première expression est évaluée comme False (fausse).

Faisons donc notre petit test de l'âge. La déclaration sera "Si l'âge est inférieur à 1, ou si l'âge est supérieur à 130, alors l'utilisateur ment, sinon dites-lui que son âge est le plus bel âge". Traduisons cette expression en code. Regardez le résultat dans le Listing 3.2.

Listing 3.2 : Exemple de programme utilisant la déclaration if

```
program ExempleIF2;

Uses
 Forms;

var
 EntreeUtilisateur : Integer;

begin
 Write ('Quel âge avez-vous ?');
  ReadLn (EntreeUtilisateur);
  If (EntreeUtilisateur < 1) or (EntreeUtilisateur > 130) then
     WriteLn ('Vous ne dites pas la vérité.")
  else
     WriteLn (EntreeUtilisateur, ' ans est le plus bel âge !');
 ReadLn
end.
```

N'oubliez pas que la première expression est constituée de tout ce qui se trouve entre les mots réservés If et Then. L'expression

```
(EntreeUtilisateur < 1) or (EntreeUtilisateur > 130)
```

doit pouvoir être évaluée comme True ou False, et c'est le cas. Si l'âge est inférieur à un, la première partie (EntreeUtilisateur<1) devient True. Si l'âge est supérieur à 130, c'est l'expression (EntreeUtilisateur>130) qui devient True. Dans les deux cas, avec le or (ou) logique entre les deux, si l'un des deux est True, l'ensemble est évalué à True. Si la première expression est évaluée comme True, la deuxième expression est exécutée. Si la première expression est False, alors la troisième expression (le WriteLn() placé après la déclaration Else) est exécutée. Vous tenez là un exemple simple mais efficace des éléments de construction des clauses conditionnelles.

Case...of

Avant l'apparition des contrôles d'éditions masqués, un programmeur qui souhaitait limiter l'entrée d'un utilisateur à certains caractères devait analyser chaque caractère séparément. Chaque fois qu'un caractère était entré, le code vérifiait ce qu'était ce caractère et décidait de la manière de le traiter. La déclaration Case s'avérait alors très utile. Cette déclaration Case permet au programmeur de faire la comparaison entre une entrée et un ensemble prédéfini de "cas" (cases en anglais), et de répondre en conséquence. Un exemple figure dans le Listing 3.3.

Listing 3.3 : Exemple de programme utilisant Case of

```
Program ExempleCase;

uses
 Forms;

var
 EntreeUtilisateur : Char;

begin
 Read (EntreeUtilisateur);
 Case EntreeUtilisateur of
   'a'        : WriteLn ('C''est un a minuscule');
   'z', 'Z' : WriteLn ('C''est un Z minuscule ou majuscule')
 else
   WriteLn ('Ce caractère n'est pas un a, un z ou un Z.');
 ReadLn
end.
```

Dans cet exemple, on demande à l'utilisateur d'entrer un caractère au clavier (la touche Entrée n'est nécessaire puisque Read ne prend que ce dont il a besoin et ne nécessite pas l'appui sur Entrée). Ensuite, l'entrée de l'utilisateur (qui se trouve maintenant dans la variable EntreeUtilisateur) est comparée avec la première constante. Si cette constante (le caractère 'a' en l'occurrence) correspond aux données de EntreeUtilisateur, la déclaration située après les deux points de la ligne est exécutée. Cette déclaration peut être un appel à une fonction ou à une procédure (nous reviendrons sur ces notions tout à l'heure), ou une série de déclarations entourée d'un end et d'un begin. Ensuite, une fois la déclaration de la ligne 'a' exécutée, le reste de la déclaration Case sera ignoré car une correspondance aura déjà été trouvée.

Si la ligne 'a' ne correspond pas, c'est la ligne suivante de la déclaration Case qui est comparée à EntreeUtilisateur. S'il y a correspondance, la ligne de déclaration associée à cette nouvelle constante est exécutée, et ainsi de suite. Si aucune correspondance n'est trouvée, la déclaration else (si elle existe) sera exécutée.

Info *L'instruction située à droite des deux points dans l'instruction* `Case` *peut être un appel de fonction, de procédure, ou un ensemble de lignes de code délimitées par un bloc* `begin .. end`*, ce qui apporte une grande souplesse à vos programmes. Notez la présence du* `end` *qui vient fermer la structure* `Case` *: il est indispensable.*

Astuce *Il est préférable de toujours utiliser une clause else dans une déclaration Case pour parer à toute éventualité.*

Quelques règles concernant Case :

- Si vous utilisez plus d'une constante sur une ligne de comparaison, celles-ci doivent être séparées par des virgules, sauf si vous indiquez un intervalle de valeurs (de 'a' à 'z' par exemple).
- L'expression ou la constante à laquelle on compare l'expression doit être d'un type ordinal de taille byte ou word. Vous ne pouvez donc pas utiliser un type String ou LongInt comme argument.
- Les constantes auxquelles vous comparez votre expression (EntreeUtilisateur dans notre exemple) ne peuvent se recouvrir. Cela signifie que la déclaration Case suivante ne fonctionnera pas :

```
Program MauvaisCase;

Uses
  Forms;

var
 EntreeUtilisateur : Char;

begin
 Read (EntreeUtilisateur);
```

```
 Case EntreeUtilisateur of
   'a'        : WriteLn ('C''est un a minuscule');
   'a', 'z', 'Z' : WriteLn ('C''est un Z, un z ou un a.');
   'Zap'      : WriteLn('Ce test est impossible à réaliser');
 else
   WriteLn ('C''est un caractère autre que a, z ou Z.');
 ReadLn
end.
```

Si vous tentez d'exécuter cette application, le compilateur vous annoncera une erreur duplicate case label. C'est une bonne chose car même si le compilateur permettait un tel programme, celui-ci n'aurait aucun sens. Pour que la déclaration Case s'exécute correctement, toutes les valeurs doivent être uniques. Un 'a' en double plongerait Delphi dans un dilemme.

On a également ajouté une nouvelle option au `Case` : `Zap`. La variable du `Case` étant de type `char`, cette valeur chaîne provoquera une erreur à la compilation. Il est impossible de mélanger différents types au sein d'une même clause `Case`.

Structures de boucle

Il arrive qu'il faille exécuter un ensemble d'instructions plusieurs fois de suite, jusqu'à ce qu'une condition spécifique soit remplie. Le Pascal Objet offre à cet effet plusieurs instructions de boucle :

- `Repeat .. Until` : Cette boucle s'exécute toujours au moins une fois.
- `While .. Do` : Cette boucle peut ne jamais être parcourue.
- `For` : Cette boucle est parcourue un nombre de fois connu d'avance.

Repeat...until

Le principe de Repeat...until (Répète...jusqu'à) est très simple. Les déclarations qui se trouvent entre le repeat et le until sont répétées jusqu'à ce que la condition définie après le mot réservé Until soit évaluée comme True (Vraie). Un exemple figure dans le Listing 3.4.

Listing 3.4 : Exemple de programme utilisant Repeat...until

```
program ExempleRepeat;

Uses
 Forms;

begin
```

```
 repeat
     Write('Entrez une valeur : ');
     Read(I);
 until (I = 'q') or (I = 'Q');
 ReadLn
end.
```

Ce programme continuera de demander des caractères à l'utilisateur jusqu'à ce qu'il appuie sur la touche 'q'. De prime abord, la syntaxe peut désorienter car ici les begin/end ne sont pas nécessaires. Le repeat et le until tiennent lieu de marqueurs qui délimitent les instructions de la boucle.

Comme vous pouvez le voir dans le listing ci-avant, le segment de code situé entre les déclarations repeat et until sera exécuté une fois avant même que la condition du until (ici (I = 'q') or (I = 'Q')) ne soit vérifiée. Vous devez vous assurer que le segment de code peut prendre en charge une itération avant de vérifier son état. Si vous n'êtes pas sûr de vous, utilisez plutôt une autre instruction de boucle. Autre point intéressant, celui du nombre d'itérations qu'effectuera cette boucle. On ne connaît pas la réponse. Cette boucle peut s'exécuter une fois ou mille, selon ce qu'entrera l'utilisateur. C'est un autre point à considérer lorsque vous utilisez cette commande.

While...Do

Le While...Do (tant que...faites) est une variante de l'instruction précédente. Cette commande a une fonction similaire, à deux différences près :

1. La déclaration conditionnelle est vérifiée *avant* d'entrer dans la boucle.
2. Si la condition est False, la boucle ne sera pas exécutée une nouvelle fois (à l'inverse de la boucle Repeat...Until).

Regardez comment obtenir le même résultat que précédemment mais avec une boucle While... Do cette fois (Listing 3.5).

Listing 3.5 : Exemple de programme utilisant `While...Do`

```
program ExempleWhileDo;

Uses
 Forms;

var
  I: Char;

begin
 While (I <> 'q') and (I <> 'Q') Do
```

```
    begin
      Write('Entrez une valeur : ');
       Read(I);
    end;
 ReadLn
end.
```

Remarquez que vous devez inverser la logique de (I = 'q') or (I = 'Q') en (I <> 'q') and (I<> 'Q') pour que la boucle. Cela est dû à un changement de perspective. Vous êtes passé de "Faites ceci jusqu'à..." à "Tant que... est vrai faites...". La logique est inversée. Comme dans la boucle Repeat, il n'y a aucun moyen de savoir à l'avance combien de fois la boucle s'exécutera. Ce nombre dépend entièrement de ce qu'entrera l'utilisateur.

For...Do

La boucle For...Do est l'une des instructions de boucle les plus simples. Vous l'utilisez lorsque vous savez combien de fois vous souhaitez que la boucle s'exécute. Le Listing 3.6 vous fournit un exemple d'utilisation.

Listing 3.6 : Exemple de programme utilisant For...Do

```
program  ExempleFor;

uses
 Forms;

var
 Compte : Integer;

begin
 For Compte := 1 to 10 do
   WriteLn ('Hello');
 ReadLn
end.
```

Dans le Listing 3.6, vous définissez une variable Compte de type Integer. La boucle For est mise en place comme suit :

- La variable Compte contiendra la valeur courante de la boucle.
- Compte aura automatiquement 1 pour valeur initiale.
- La déclaration qui figure après le do est exécutée une fois (si cette déclaration est un begin, tout le code qui suit est exécuté jusqu'à ce qu'un end apparaisse).
- Compte est incrémenté de un (on ajoute un à Compte).

- Si Compte est supérieur à la valeur de terminaison (10), la boucle s'achève et l'exécution se poursuit avec la déclaration qui se trouve *après* la boucle For.
- Si Compte n'est pas supérieur à la valeur de terminaison (10), alors la déclaration qui se trouve après le do est exécutée à nouveau.

  Le terme bloc de code recouvre de façon générique plusieurs situations sous Delphi. Au départ, un bloc de code est composé de l'ensemble des instructions présentes entre un `begin` et un `end`. Dans un sens plus large, un bloc de code désigne l'ensemble des instructions qui apparaissent dans une procédure ou dans une fonction.

  Lorsque vous utilisez la boucle `For... Do`, gardez à l'esprit les règles suivantes :

- Vous devez bien avoir en tête quelques règles. La variable que vous utilisez pour contenir la valeur courante de la boucle (Compte dans cet exemple) doit être à portée (nous reviendrons sur cette notion par la suite). Pour dire les choses simplement, cela signifie que Compte doit être une variable locale valide pour le bloc dans lequel la boucle For se trouve.
- La valeur initiale (1) doit être inférieure à la valeur finale (10). C'est logique puisque le For est incrémental. Vous ne pouvez pas utiliser Compte comme paramètre formel pour une procédure. Vous ne pouvez pas modifier la valeur de Compte au sein de la boucle For elle-même.
- La variable de contrôle Compte doit être d'un type compatible avec les valeurs initiale et finale (1 et 10). Delphi 3 doit pouvoir placer ces valeurs, et toutes les valeurs intermédiaires, dans Compte.
- Vous devez aussi vous souvenir que Compte n'est plus défini une fois que le point d'exécution quitte la boucle, il est donc impossible d'utiliser sa valeur courante pour quoi que ce soit en dehors de la boucle.

  Il n'existe qu'une variation de la boucle For, que nous allons voir à présent. Vous pouvez décrémenter la variable Compte en utilisant cette variante. Le Listing 3.7 vous fournit un exemple.

Listing 3.7 : Autre exemple de programme utilisant For...Do

```
program  ExempleFor2;

uses
 Forms;

var
 Compte : Integer;

begin
 For Compte = 10 downto 1 do
   WriteLn ('Hello');
 ReadLn
end.
```

Ce programme ressemble au programme précédent, à ceci près que le mot réservé to a été remplacé par downto et que les valeurs initiale et finale ont été inversées. La boucle For fonctionne de la même façon qu'auparavant sauf qu'ici Compte est décrémenté de 10 à 1, avant de s'achever.

Branchements

Une autre possibilité d'un langage tel que le Pascal Objet est la capacité d'effectuer un *branchement* dans le code. En d'autres termes, il s'agit de la possibilité de se rendre à un autre endroit du code si nécessaire. Plusieurs mécanismes existent à cet effet :

- `Goto`
- `Break`
- `Continue`
- `Exit`
- `Halt`
- `RunError`

Goto

La déclaration Goto vous permet de vous déplacer de votre position actuelle dans le code jusqu'à une ligne spécifiquement étiquetée. Une telle étiquette ou *label* est définie en utilisant le mot réservé `label` pour définir un nouveau nom de label. Ce nom de label peut alors être affecté à une ligne de code en écrivant label, deux points (:), puis la ligne du code exécutable. Un exemple figure dans le Listing 3.8.

Listing 3.8 : Exemple de programme utilisant Goto

```
label Ici;
Program ExempleGoto

uses
 Forms;

var
Reponse : String;

label Ici;

begin
  Ici : WriteLn ('Hello');
```

```
  WriteLn ('Voulez-vous exécuter une nouvelle fois ?');
  ReadLn (Reponse);
  If (Reponse = 'o') then
     Goto Ici;
  ReadLn
end.
```

Vous avez défini un label nommé Ici que vous allez utiliser dans votre code. Le label est défini en dehors de la zone de code exécutable, mais après les définitions const, var, type, etc. La ligne 'Hello' sera toujours exécutée. Le WriteLn demande à l'utilisateur s'il désire relancer le programme. Si la réponse est 'o'; alors Goto Ici est exécuté. Sinon, le dernier ReadLn est exécuté et lorsque l'utilisateur appuie sur Entrée, le programme se termine.

Info

Le concept de label est très ancien et a été très souvent utilisé. Il y a certainement bien d'autres moyens d'écrire cet exemple sans avoir besoin d'y faire figurer le moindre Goto. Et c'est là la pomme de discorde qui a lancé une polémique qui fait encore rage. Nombreux sont les développeurs qui pensent que l'usage de Goto trahit en fait de mauvaises habitudes de programmation, et qu'un examen plus poussé du code permettrait de trouver une façon plus élégante de faire les choses. D'autres, plus pragmatiques, pensent que si l'outil existe, autant l'utiliser. A vous de décider !

Break

Imaginons que vous soyez au beau milieu d'une boucle For ou While et que vous trouviez une condition qui exige impérativement que vous sortiez de cette boucle. Sortir d'une boucle de façon impromptue n'a rien d'évident, et c'est pourquoi la déclaration Break vous permet de terminer complètement l'exécution d'une boucle. Le Listing 3.9 vous montre un exemple d'utilisation. Nous allons réécrire ExempleWhileDo pour la circonstance.

Listing 3.9 : Exemple de programme utilisant Break

```
program ExempleBreak;

Uses
 Forms;

var
  I: Char;

begin
 I := ' ';
 While True Do
    begin
```

```
      Write('Entrez une valeur :');
       Read(I);
       If (I = 'q') or (I = 'Q') then
         Break;
    end;
  {Le break fera cesser l'exécution ici}
  ReadLn
end.
```

Ce programme peut sembler étrange mais il a le mérite d'illustrer un point intéressant. La boucle While continuera indéfiniment car l'expression est toujours évaluée comme True. La seule façon de sortir de cette boucle consiste à utiliser la déclaration Break. Ainsi, après avoir lu l'entrée de l'utilisateur dans le Read, la déclaration If détermine si la touche appuyée était un q minuscule ou majuscule. Si c'est bien le cas, le Break est alors invoqué. L'exécution s'arrête alors ici pour ce qui concerne la boucle, et le reste de l'application est ensuite exécuté.

Continue

La condition inverse existe aussi, lorsque vous ne souhaitez pas sortir de la boucle, mais simplement terminer ce que vous êtes en train de faire dans l'itération en cours pour passer à l'itération suivante. La commande Continue vous permet de faire cela. Lorsqu'elle est utilisée dans une boucle For, Repeat ou While, la déclaration Continue stoppe le traitement dans l'itération en cours et rend le contrôle à la boucle elle-même pour qu'elle passe à l'itération suivante. Le Listing 3.10 fournit un exemple d'utilisation.

Listing 3.10 : Exemple d'utilisation de Continue

```
program ExempleContinue;

Uses
 Forms;

var
  I: Char;
   Compte : Integer;

begin
 I := ' ';
 For Compte = 1 to 100 Do
    begin
      Write('Entrez une valeur : ');
       Read(I);
       If (I = 'q') or (I = 'Q') then
```

```
          Continue;
        WriteLn (' Ceci ne sera exécuté que si l''entrée de l''utilisateur
          n''est pas un q ou un Q.')
    end;
 ReadLn
end.
```

Dans ce programme, votre boucle For s'exécute 100 fois. A l'intérieur de cette boucle, vous scrutez les touches sur lesquelles l'utilisateur appuie. Si cette touche est un q, la déclaration Continue est alors invoquée et l'exécution revient à la ligne For qui incrémente d'un la variable Compte avant de reprendre. Il s'agit là d'un mécanisme supplémentaire dont vous pouvez tirer parti pour arriver à vos fins.

Info

Une fois la variable de contrôle (Compte) incrémentée, elle est également testée. Vous pouvez utiliser les commandes Break et Continue si vous rencontrez une condition d'erreur dans votre boucle ; utilisez Continue pour passer à l'élément suivant ou Break pour ne pas traiter du tout les éléments qui suivent.

Exit

Le nom indique parfaitement la fonction de cette déclaration. Exit permet de sortir du bloc de code courant. Si ce bloc est le programme principal, Exit entraîne la fin de celui-ci. Si le bloc courant est imbriqué, Exit fait continuer le programme avec le bloc extérieur suivant qui passe à la déclaration qui vient immédiatement après la déclaration ayant passé le contrôle au bloc imbriqué. Si le bloc courant est une procédure ou une fonction (nous reviendrons sur ce point dans la section consacrée aux programmes), Exit fait en sorte que le bloc appelant continue en reprenant à la déclaration qui suit immédiatement l'appel au bloc. Le Listing 3.11 donne un exemple d'utilisation.

Listing 3.11 : Exemple de programme utilisant Exit

```
program ExempleExit;

Uses
 Forms;

var
  I: Char;
   Compte : Integer;

begin
   repeat
      Write('Entrez une valeur : ');
```

```
        Read(I);
        If  I := 'Q' then
           Exit;
    Until False;
    ReadLn
end.
```

Dans notre exemple, vous avez créé une boucle infinie en définissant la portion Until de la boucle comme étant False. La seule façon de terminer la boucle est que l'utilisateur appuie sur la touche Q. A ce moment-là, le If...Then est évalué comme True et la commande Exit entre en action. Comme le bloc de code dans lequel vous vous trouvez est le programme principal, le programme s'arrête.

Halt

Halt permet de mettre fin à votre application à votre guise. Là où vous placez la command Halt, l'application se termine. Avant de placer cette commande dans votre application, vous devez cependant vous poser quelques questions :

- Avez-vous laissé une base de données ou un fichier ouverts ? Si vous arrêtez brusquement ils risquent alors d'être endommagés.
- Avez-vous alloué un espace mémoire que vous n'avez pas encore libéré ? Dans ce cas vous gâcherez de la mémoire en quittant maintenant.

On ne saurait trop insister : soyez très prudent dans l'usage de cette commande. Elle est très puissante et, mal employée, elle pourrait vous causer des ennuis. Cette commande peut par exemple être utilisée lorsque l'application rencontre une erreur critique. Certains des codes de retour provenant d'appels Windows ou Delphi vous indiquent que quelque chose d'imprévu et de désagréable vient d'arriver. Peut-être des problèmes liés à la mémoire ou au disque. Vous pouvez en ce cas mettre un terme brutal à l'application, plutôt que d'en sortir en y mettant les formes. Cependant, il vaut toujours mieux essayer la manière élégante. La commande Halt ne doit être utilisée qu'en tout dernier ressort. Si vous arrêtez ainsi votre application, elle risque de laisser ouverts des fichiers, de ne pas libérer de mémoire et de laisser en suspens et à la dérive de nombreux éléments de votre système. Le Listing 3.12 illustre un exemple d'utilisation.

Listing 3.12 : Exemple de programme utilisant Halt

```
program ExempleHalt;

Uses
 Forms;

var
  I: Char;
  Compte : Integer;
```

```
begin
  repeat
     Write('Entrez une valeur : ');
      Read(I)
      If  I := 'Q' then
         Halt;
  Until False;
end.
```

Dans le Listing 4.13, lorsque l'utilisateur appuie sur la touche Q, le programme s'arrête instantanément. Ce n'est pas une façon très harmonieuse de dire au revoir, mais elle a le mérite d'être efficace. Dans des circonstances normales (et pour une application plus complexe), il est préférable d'exécuter avant la sortie une portion de code chargée de nettoyer l'environnement.

RunError

Voilà une belle commande. Si vous n'aimez pas les codes d'erreur que Delphi 3 vous assène, vous pouvez créer les vôtres. A n'importe quel moment de l'exécution d'un programme, vous pouvez invoquer la commande RunError avec un paramètre Integer, et l'exécution du programme sera arrêtée et le code d'erreur que vous avez entré sera indiqué comme raison de l'échec du programme. Le Listing 3.13 illustre un exemple d'utilisation.

Listing 3.13 : Exemple de programme utilisant RunError

```
program ExempleRunError;

Uses
 Forms;

var
  I: Char;
  Compte : Integer;

begin
  repeat
     Write('Entrez une valeur : ');
      Read(I)
      If  I := 'Q' then
         RunError (240);
  Until False;
  ReadLn
end.
```

Ce programme a été écrit pour générer une erreur d'exécution 240 (une erreur inventée pour la circonstance) chaque fois que l'utilisateur appuie sur la touche Q. Si le debogueur est en train de tourner, il est possible de voir venir cette erreur d'exécution. Cette commande est particulièrement pratique si vous trouvez que votre application ne génère pas assez d'erreurs à votre goût... Les erreurs d'exécution sont reprises dans la documentation en ligne de Delphi. A ce sujet, vous remarquerez que l'erreur 240 n'y figure pas : elle est définissable par l'utilisateur. Les codes d'erreurs ne peuvent pas dépasser 255.

Programmes

Tout au long de ces premières leçons, vous avez passé une sorte de baptême du feu des structures d'une application Delphi 3. Nous parlerons en détail au Jour 5 des fichiers qui constituent votre projet et de la façon de les gérer, mais il est maintenant nécessaire de parler de votre application du point de vue de la structure interne. Vous avez pu voir (et essayer) les programmes d'exemples qui ont émaillé jusqu'ici le cours de notre discussion. Nous allons disséquer l'un d'entre eux.

Toutes les applications Delphi 3 ont ceci de commun qu'elles contiennent toutes un segment de code qui est le point principal autour duquel tout le reste s'articule. Ce segment de code commence par le mot Program. C'est là que commence l'exécution de votre application. La clause Uses se trouve généralement après. Cette section permet d'inclure d'autres unités de code (nous reviendrons sur ce sujet dans la section consacrée aux *Unités*). Après la clause Uses vient la section de déclaration. Les mots réservés const, type et var sont les déclarations que l'on trouve habituellement dans cette section. Ici, sont définis les types de données globaux, les variables et les constantes. Enfin, vous trouvez la paire begin...end, avec un point situé après la déclaration end. Ce end suivi d'un point indique la fin du code exécutable, et il ne peut y avoir qu'un seul end. dans l'exécutable principal. Le seul endroit où un end. peut faire son apparition est dans une unité ou un DLL.

Un programme a donc l'aspect suivant (voir Listing 3.14).

Listing 3.14 : Exemple de programme principal

```
Program ExempleProg;

Uses
  Forms;

const
 Impots = 7.75;
 Mort = True;
```

```
var
 Salarie : String;

begin
 If (Impots > 0) and Mort then
   Salarie := 'Moi';
 WriteLn ('R.I.P.')
 ReadLn

end.
```

Comme vous pouvez le voir, il n'y a que deux constantes, la mort et les impôts. Tout ce qui se trouve entre le begin et le end est exécuté dans l'ordre de haut en bas, et le programme s'arrête lorsqu'il rencontre le end suivi d'un point. Jusqu'ici, rien de plus simple.

Procédures

Je tiens la procédure pour la plus belle invention depuis l'ampoule électrique. Ou peu s'en faut. Les procédures vous permettent d'améliorer sensiblement la qualité de votre vie de programmeur. Voyons ensemble comment les procédures peuvent améliorer la productivité, la clarté et comment elles réduisent les bogues dans votre code.

Lorsque vous êtes plongé dans la programmation, les choses les plus évidentes vous échappent parfois. Si vous devez écrire un programme qui affiche un damier, vous pouvez être tenté de procéder ainsi :

Listing 3.15 : AfficheDamier utilisant des WriteLn

```
program AfficheDamier;

Uses
 Forms;

begin
 WriteLn ('    ¦   ¦    ');
 WriteLn ('--------- ');
 WriteLn ('    ¦   ¦    ');
 WriteLn ('--------- ');
 WriteLn ('    ¦    ¦   ')
end.
```

Info

A cause des polices proportionnelles utilisées par Windows, l'affichage de ce plateau de jeu risque d'être relativement illisible. Nous n'avons pas recherché ici l'esthétique, notre propos étant purement illustratif.

Ce programme fonctionne (même si le damier n'est pas fabuleux) mais n'a rien d'élégant. Vous pouvez voir comme il se répète ? Vous faites plusieurs fois la même chose. C'est ici qu'intervient la notion de procédure. Le Listing 3.16 nous montre le programme revu et corrigé.

Listing 3.16 : AfficheDamier utilisant des procédures

```
program AfficheDamier;

Uses
 Forms;

{normalement c'est là que l'on placerait les déclarations Uses, Const, Var,
Type et autres.}

   procedure Verticales;
     begin
       WriteLn ('    ¦   ¦    ');
     end;

   procedure Horizontales;
     begin
       WriteLn ('------------');
     end;

begin
 Verticales;
 Horizontales;
 Verticales;
 Horizontales;
 Verticales;
end.
```

Dans cet exemple, vous prenez une section (en l'occurrence, une ligne) de code que vous utilisez souvent pour la placer dans un emballage, vous donnez un nom à cet emballage, puis vous utilisez ce nom pour accéder au code qui se trouve dans l'emballage, et ce autant de fois que vous le désirez. Pour ce faire, vous avez créé deux procédures, une qui affiche les lignes horizontales et l'autre les lignes verticales. Remarquez que les procédures ont été déclarées entre le mot réservé program et le début du code exécutable (qui est la déclaration begin).

La première chose à faire lorsque vous créez une procédure est de créer un en-tête. L'en-tête est composé du mot réservé procedure et d'un nom de procédure. Dans notre exemple procedure Horizontales; est l'en-tête de procédure. L'en-tête est très similaire à la ligne program AfficheDamier de votre programme principal. Les similarités entre le programme principal et la procédure ne s'arrêtent d'ailleurs pas là. La procédure est une sorte de mini-programme et son format est le même. Après l'en-tête, vous pouvez trouver une section contenant Const, Type et Var, ou même une autre procédure. Une procédure peut en comprendre une imbriquée. Enfin, la procédure comporte une section exécutable marquée par la paire begin...end;.

Maintenant que vous avez créé cette procédure Horizontales, vous pouvez appeler cette procédure dans votre code. Lorsque vous appelez ce code en utilisant le nom qui lui est propre, le contrôle passe à cette procédure. A la fin de l'exécution de la procédure, celle-ci redonne le contrôle à la ligne de code qui se trouve *après* la ligne d'appel à la procédure.

Les procédures permettent de modulariser le code et permettent la réutilisation. Un autre avantage des procédures est la fiabilité du code ainsi généré. Si vous écrivez six fois de suite le même morceau de code dans votre application, le risque d'erreur est multiplié par six. Par contre, si vous utilisez une procédure, vous écrivez le code une fois, le testez une fois, et l'utilisez ensuite autant de fois que vous le souhaitez. Les risques d'erreurs sont moindres, et la qualité du code est améliorée.

Passage de paramètres

L'intérêt des procédures serait limité si leur utilité ne dépassait pas le cadre de l'exemple ci-avant. Une procédure qui affiche une ligne de traits n'est certes pas d'une folle utilité. Pour rendre les procédures plus précieuses, il faut les rendre plus souples. L'idéal serait de transmettre des données supplémentaires à la procédure, au moment même où vous l'appelez, pour lui donner un sens et un but particulier. C'est là le rôle des paramètres. Réécrivons la procédure Verticales pour inclure un paramètre permettant de varier le nombre de fois où les lignes sont imprimées.

Listing 3.17 : Procédure Horizontales

```
procedure Horizontales (NombreDeFois : Integer);

var
 Compte : Integer;

    begin
      for Compte := 1 to NombreDeFois do
       WriteLn ('-----------------');
     end;
```

Cette procédure a gagné en souplesse. Vous avez introduit une variable NombreDeFois qui est transmise à la procédure. Le code (NombreDeFois : Integer); est appelé liste de paramètres formels. Il s'agit d'une liste indiquant les données (et le type de ces données) qui doivent être transmises à la procédure lors de son appel. Dans notre exemple, vous souhaitez transmettre une variable Integer dans la procédure pour contrôler le nombre d'exécution du WriteLn. En donnant un nom à cette variable, vous effectuez en fait une déclaration de variable ailleurs que dans la section var de la procédure. A présent, la procédure comporte une variable NombreDeFois de type Integer qui peut être utilisée en n'importe quel endroit de cette procédure.

Cependant, comme cette variable a été définie dans la procédure même, elle ne peut être utilisée que là (reportez-vous aux paragraphes consacrés à la notion de *portée* dans la suite de cette section). Comme vous pouvez le voir, vous utilisez ensuite NombreDeFois comme valeur finale de la boucle for qui exécute le WriteLn le nombre de fois voulu. La procédure est devenue beaucoup plus souple.

Maintenant que vous avez créé une liste de paramètres formels dans la déclaration de la procédure, vous devez modifier votre façon d'appeler la procédure. Vous ne pouvez plus vous contenter de dire Horizontales. Pour chaque paramètre formel que vous identifiez, vous devez faire figurer un paramètre correspondant dans l'appel. Il est *nécessaire* qu'il y ait correspondance parfaite entre les paramètres formels et réels. Votre en-tête de procédure était

```
procédure Horizontales (NombreDeFois : Integer);
```

Vous devez donc transmettre un nombre Integer à cette procédure lorsque vous l'appelez. Le nouveau code du programme se trouve dans le Listing 3.18.

Listing 3.18 : Exemple de programme utilisant des paramètres

```
program ExempleParam;

Uses
 Forms;

var
 Nombre : Integer;

  procedure Horizontales (NombreDeFois : Integer);

  var
    Compte : Integer;

    begin
      for Compte := 1 to NombreDeFois do
       WriteLn ('-----------------');
     end; {Procedure Horizontales}

```

```
begin
  WriteLn ('Combien de lignes horizontales souhaitez-vous imprimer ?');
  ReadLn(Nombre);
  Horizontales(Nombre);
  ReadLn
end.
```

Vous avez peut-être remarqué que lorsque vous appelez Horizontales, vous lui transmettez la variable Nombre. Il s'agit d'un paramètre. Le contenu de Nombre est copié dans la procédure Horizontales et se retrouve dans la variable NombreDeFois qui est définie localement. La procédure dispose donc des données de la variable globale Nombre dans NombreDeFois, et peut commencer ses manipulations. A la fin de Verticales, les données de NombreDeFois disparaissent, car la variable locale est détruite à la sortie de la procédure.

Lorsque vous transmettez (ou *passez*) plusieurs paramètres à une procédure, l'ordre dans lequel vous le faites est primordial. La seule façon pour Delphi 3 de dire si les paramètres formels et réels correspondent consiste à regarder l'ordre dans lequel ces paramètres sont passés. Prenons un autre exemple d'utilisation des paramètres.

```
Procedure Truc (Nombre : Integer; Machin : String);

begin
  WriteLn ('Le nombre est ', Nombre);
  WriteLn ('La chaine est ', Machin)
end;
```

Cette procédure prend deux paramètres, l'un est une valeur Integer, l'autre une valeur String. Lorsque vous appelez cette procédure, vous devez effectuer l'appel de procédure lui-même et passer à la procédure un Integer et une String, dans cet ordre. Sinon, le programme ne se compilera pas. L'appel de cette procédure est illustré dans le Listing 3.19.

Listing 3.19 : Le programme Truc

```
program ExempleTruc;

Uses
  Forms;

var
 NombreUtilisateur : Integer;
 ChaineUtilisateur : String;

Procedure Truc (Nombre : Integer; Machin : String);

```

```
begin
  WriteLn ('Le nombre est ', Nombre);
  WriteLn ('La chaîne est ', Machin)
end;

begin
 Write ('Entrez votre chaîne :');
 ReadLn (ChaineUtilisateur);
 Write ('Entrez votre nombre :');
  ReadLn (NombreUtilisateur);
  Truc (NombreUtilisateur, ChaineUtilisateur)
 ReadLn
end.
```

Comme vous pouvez le voir, vous passez deux paramètres à Truc, en vous assurant que vous les passez dans l'ordre adéquat. Vous pouvez également passer à la fonction des données littérales (plutôt que des variables). Ainsi, vous pourriez appeler Truc (3,'Bonjour tout le monde'); qui transmettrait ces données littérales dans les variables locales pour qu'elles y soient traitées.

Visibilité et portée

A mesure que vos applications deviendront de plus en plus fragmentées lors du recours à des procédures et fonctions, vous aurez besoin de savoir avec certitude quel sera le destin de vos variables. Avec l'arrivée des procédures, des fonctions et des unités, les risques de recouvrement de variables se font plus grands. Il est donc temps de parler de la visibilité et de la portée. Créons un programme d'exemple (Listing 3.20) et intéressons-nous à la façon dont les variables sont perçues.

Listing 3.20 : ExempleVisible

```
program ExempleVisible;

Uses
 Forms;

var
 A : Integer;

  procedure Exterieure;

     var
      B : Integer;
```

```
      procedure Interieure;

      var
        C : Integer;

       begin
       C := 3;
       B := 8;
       A := 4; {Je peux voir A à partir des déclarations du programme
    principal}
       end; {de la procédure Interieure}

    begin
       B := 5;
       C := 5; {ceci n'est pas valide, je ne peux voir que vers l'extérieur,
    et pas vers l'interieur}
       A := 9 {Je peux voir A à partir des déclarations du programme
    principal }
    end; {de la procédure Exterieure}

  procedure UneAutre;

    var
       D : Integer;

   begin
     D := 9;
     A := 55; {Je peux voir A à partir des déclarations du programme
    principal}
     B := 4; {Ceci serait illégal, je peux voir vers l'extérieur, mais pas
    vers l'interieur }
     C := 5; { Ceci serait illégal, je peux voir vers l'extérieur, mais pas
    vers l'interieur }
   end; {de la procédure UneAutre }

begin
   A:= 1
    {Je ne peux pas référencer une des variables locales dans une procédure
    ici. Aucune n'est visible.}
end.
```

Dans cet exemple, votre application contient une procédure imbriquée. Il faut garder à l'esprit ces deux règles qui indiquent quelles sont les variables que peuvent voir chacune des parties de vos programmes :

- Règle 1 : Les variables ne sont visibles et disponibles que dans le bloc où elles sont définies.
- Règle 2 : Dans les procédures ou fonctions imbriquées, vous pouvez toujours voir de l'intérieur vers l'extérieur. Cela signifie que les procédures qui sont imbriquées au niveau le plus bas peuvent voir les variables définies dans leurs procédures parent, et dans les procédures parent de celles-ci, et ainsi de suite.

Info *On peut avoir l'impression que la règle 2 contredit la règle 1. Ce n'est pas le cas. Revenons à la définition d'un bloc. Comme une procédure imbriquée est définie dans le bloc de code de son parent, la visibilité est là, et si ce parent est défini dans le bloc de code de son propre parent, la procédure imbriquée au niveau le plus bas peut voir jusqu'au sommet de la hiérarchie. Vous devez mémoriser ceci : vous pouvez toujours voir vers l'extérieur, mais jamais vers l'intérieur.*

En appliquant ces règles au Listing 3.20, nous pouvons tirer quelques conclusions :

- Comme la variable C a été créée dans la procédure Interieure, elle n'est visible que dans cette procédure.
- Comme la variable B a été définie dans la procédure Exterieure, elle est visible dans cette procédure, mais aussi dans la procédure Interieure, qui peut voir au-dessus d'elle (au niveau d'imbrication supérieur si vous préférez).
- Comme la variable A a été définie dans le corps principal du programme, elle est visible pour toutes les procédures et fonctions du corps principal. C'est le parent de toutes les procédures enfant. En revanche, les variables des procédures enfant ne sont pas visibles pour le corps principal, car ce dernier ne peut regarder que plus "haut", et pas plus "bas".
- Comme la variable D a été définie dans la procédure UneAutre, elle n'est visible que dans UneAutre, puisque UneAutre ne contient pas de procédures enfant imbriquées. La variable A est visible dans UneAutre car c'est la variable de son parent.

En plus de la notion de visibilité, vous devez prendre en considération celle de *portée*, c'est-à-dire l'étendue de programme dans lequel il est valide d'utiliser une variable. La règle est simple. Une variable ne dépasse pas le bloc dans lequel elle est définie. La portée de la variable C ne dépasse pas la procédure Intérieure. Elle n'a pas d'existence en dehors de cette procédure. Des affectations à la variable C effectuées en dehors de la procédure Intérieure (dans la procédure Exterieure par exemple), vous vaudraient une erreur de compilation, indiquant que la variable C n'est pas définie (dans la procédure Exterieure).

Nous venons de mettre en lumière la différence qui existe entre des variables locales et globales. Les variables globales sont définies dans le corps du programme principal et sont visibles dans toute l'application. Les variables locales sont définies dans des procédures et des fonctions, et ne sont visibles que dans certaines portions de l'application, selon les divers degrés d'imbrication de la procédure.

Fonctions

Les procédures ne sont pas la seule structure à votre disposition en Pascal Objet. Les procédures jouent certes un rôle prépondérant dans la programmation Delphi, mais elles ne peuvent répondre à toutes les exigences. Prenons un exemple. Imaginons que le monde ne soit rempli que de procédures. Essayer de faire un peu de calcul prendrait des proportions hallucinantes dans un tel monde. Si vous aviez une procédure pour élever au carré, elle ressemblerait à ceci :

```
procedure Carre (Le_Nombre : Real; Var Le_Resultat);

 begin
   Le_Resultat := Le_Nombre * Le_Nombre;
 end;
```

Pour utiliser cette procédure, vous l'appelleriez comme le montre le Listing 3.21 :

Listing 3.21 : Exemple de programme n'utilisant PAS de fonction

```
program ExempleFonction;

Uses
 Forms;

  procedure Carre (LeNombre : Real; Var LeResultat : Real);

    begin
      LeResultat := LeNombre * LeNombre
    end;

var
 EntreeUtilisateur , LaReponse : Real;

begin
 Write ('Entrez le nombre à élever au carré :');
 ReadLn (EntreeUtilisateur);
 Carre (EntreeUtilisateur, LaReponse);    {Là nous avons appelé notre
   procédure}
 WriteLn (EntreeUtilisateur, ' au carre egal ', LaReponse)
 ReadLn
end.
```

Info

Remarquez la présence du Var avant LeRésultat dans la liste des paramètres formels de l'appel de fonction Carre. Il indique que vous désirez effectuer un passage par référence, c'est-à-dire tramsmettre la variable et pas simplement une copie de son contenu. Dès lors, si vous modifiez dans la procédure le contenu de la variable, cette modification sera répercutée dans le code appelant.

Ce programme fonctionne correctement, mais vous êtes obligé de créer une variable LaReponse, simplement pour contenir le résultat de la mise au carré, alors que vous n'avez besoin que d'afficher le résultat sitôt le calcul fait. Il existe une solution plus simple, qui utilise une fonction.

Une fonction est similaire à une procédure en ce sens que dans les deux cas, vous passez des paramètres. La différence est qu'une fonction est conçue pour ne renvoyer qu'une valeur. Rien de sensationnel jusqu'ici puisque vous avez déjà créé des procédures qui elles aussi renvoyaient une unique valeur. La différence ici est que la fonction renvoie cette valeur unique dans le nom de la fonction lui-même. Le nom de la fonction devient une variable temporaire qui stocke le résultat de la fonction pour le passer au code appelant.

Prenons un exemple pour illustrer ce point. En Delphi 3, il existe une fonction Sqr() qui permet de calculer le carré d'un nombre. Le Listing 3.22 montre une utilisation de cette fonction. Nous écrirons ensuite notre propre fonction.

Listing 3.22 : Exemple de programme utilisant une fonction

```
Program ExempleFonction2;

Uses
 Forms;

var
  EntreeUtilisateur : Real;

begin
  Write ('Entrez le nombre à élever au carré :');
  ReadLn (EntreeUtilisateur);
  WriteLn (EntreeUtilisateur, ' au carre égal ', Sqr(EntreeUtilisateur))
end.
```

Comme vous pouvez le voir, le Listing 4.23 utilise la fonction Sqr() et calcule le résultat de l'utilisateur dans la ligne qui affiche la réponse. Une procédure ne ferait pas l'affaire ici pour deux raisons. Vous ne disposez pas d'un endroit pour placer le résultat venant d'une procédure, et vous ne pouvez pas appeler une procédure à l'intérieur d'une déclaration d'appel à une autre procédure. Réécrivons la procédure Carre() pour en faire une fonction, et regardons les différences :

```
Function Carre (Entree : Real) : Real;

 begin
    Result := Entree * Entree
 end;
```

Les différences entre l'ancienne procédure et cette fonction sont au nombre de deux. La première se trouve dans la ligne de définition de la fonction. Vous passez l'entrée de l'utilisateur dans la fonction au moyen du paramètre Entree. Vous renverrez le résultat de la fonction au moyen du nom de la fonction, qui lui sera associé, comme n'importe quelle autre variable. Le : Real qui se trouve à la fin de la définition de la fonction définit le type de la variable Result (résultat) de la fonction. Remarquer que dans la fonction elle-même se trouve la ligne :

```
Carre := Entree * Entree
```

Pour que votre fonction puisse s'achever, vous devez affecter une valeur à la variable Result. Ce nom de variable spécial est utilisé pour transmettre le résultat de votre fonction au code appelant. Vous devez définir Result comme égal à une valeur de type Real (souvenez-vous du : Real) avant que la fonction ne s'achève. Ensuite, lorsque le contrôle revient au programme qui a appelé la fonction, le nom de fonction Carre contient en fait le résultat de la fonction. Tout cela ne dure pas puisque le nom de fonction Carre ne reste une variable temporaire que dans la ligne de code qui a appelé la fonction.

Une fois cette ligne de code exécutée, le nom Carre ne contient plus de valeur et référence un appel de fonction qui nécessite des paramètres. Réécrivons le programme d'exemple pour voir la nature toute temporaire de l'appel de fonction (voir Listing 3.23).

Listing 3.23 : Exemple de programme utilisant une fonction (3)

```
Program ExempleFonction3;

Uses
 Forms;

   Function Carre (Entree : Real) : Real;

     begin
        Carre := Entree * Entree
     end;

var
  EntreeUtilisateur : Real;

begin
  Write ('Entrez le nombre à éléver au carré :');
```

```
  ReadLn (EntreeUtilisateur);
  WriteLn (EntreeUtilisateur, ' au carre égal ', Carre(EntreeUtilisateur));
  {Si nous essayons d'afficher la valeur de Carre après la ligne qui l'a
    appelé nous provoquerons une erreur de compilation et le programme ne
    s'exécutera pas.}
  WriteLn ('Au cas où vous regardiez ailleurs, la valeur du carré est',
    Carre)
end.
```

Le deuxième WriteLn du listing ci-avant ne sera pas compilé. Le mot Carre ne comporte pas de valeur à ce moment et ne signifie plus que le nom d'une fonction. Le compilateur vous indique qu'il vous manque des paramètres pour cet appel de fonction. Sans diminuer l'utilité des fonctions, cela limite un peu leur utilisation. Vous aurez l'occasion de rencontrer de nombreuses fonctions dans Delphi 3. Les fonctions sont des outils très précieux lorsqu'il s'agit de faire des calculs, du graphisme et d'autres choses encore.

Unités

Une des raisons pour lesquelles le génie logiciel a progressé si lentement dans les temps anciens de l'informatique (il y a bien 10 ans de cela), était que chacun s'acharnait à réinventer la roue chaque fois qu'il fallait développer une nouvelle application. On ne compte plus le nombre de fois où des programmeurs ont écrit des routines de tri à bulles. Cependant, avec l'arrivée de nouveaux outils de développement, une idée se fit lentement jour.

La création de l'une unité permit au programmeur d'écrire et de compiler sa routine de tri à bulles dans une unité de code. Cette unité pouvait ensuite être réutilisée et distribuée à d'autres développeurs. Comme l'unité est compilée, les autres développeurs peuvent voir le code sans voir la source et le secret des algorithmes est précieusement gardé.

Format d'une unité

L'unité est construite comme un programme principal Delphi. La forme générale d'une unité est la suivante :

```
Unit LeNomIci;

 interface
     Uses ….

     const …
     type …
     var …
     procedure …
     function …
```

```
implementation
    Uses ….
    Label …
    const …
    type …
    var …
    procedure …
    function …

initialization {facultatif}

finalization {facultatif}

end. {Fin de l'unité}
```

La section interface de l'unité vient d'abord. C'est là que vous définissez les variables, constantes, types ou autres objets que vous souhaitez rendre disponibles au projet ou aux autres unités qui ont inclus dans leur déclaration le nom de votre unité. Cela vous permet d'inclure des structures prédéfinies qui aident le développeur à utiliser votre unité. On place ensuite dans la section interface les en-têtes de toutes les procédures et fonctions mises en œuvre dans l'unité. C'est de cette manière que Delphi 3 sait ce qui est disponible pour l'application dans votre unité.

Maintenant que vous avez rendu vos intentions publiques (dans la section interface), vous implémentez dans la section adéquate les fonctions et les procédures que vous avez décrites dans la section précédente. Là, vous pouvez tranquillement donner la mesure de votre talent en écrivant votre algorithme de chiffrement ultra-secret qui vous ouvrira les portes de la gloire (et de la prison en France, mais c'est une autre histoire).

Dans la section d'implémentation, vous placez les variables, constantes, etc. que les procédures et fonctions de cette section utiliseront. Vous pouvez également créer des procédures et des fonctions qui seront utilisées localement par les procédures et fonctions spécifiées dans la section interface. Enfin, vous implémentez ces fonctions et procédures décrites dans la section interface. La liste des paramètres doit correspondre parfaitement, ou l'unité ne sera pas compilée.

Deux autres sections de l'unité méritent toute votre attention. La première est la section initialization. Vous pouvez y définir des variables et autres, tout comme dans la section interface. Le problème est que comme la section d'interface ne contient pas de zone d'exécutable, vous ne pouvez pas initialiser ces variables en leur affectant une valeur. La section d'initialisation vous permet de le faire. Là, vous pouvez initialiser vos variables, structures d'enregistrement, variables de fichier et de manière générale tout ce qui peut avoir une valeur initiale. Ceci vous permet de tout mettre en place. Vous pouvez également initialiser les variables dans le bloc begin...end qui se trouve à la fin de l'unité. Vous pouvez y définir des variables et autres, tout comme dans la section interface. Le problème est que comme la section d'interface ne contient pas de zone d'exécutable, vous ne pouvez pas initialiser ces variables en leur affectant une valeur. La section d'initialisation vous permet de le faire. Là, vous pouvez initialiser vos varia-

bles, structures d'enregistrement, variables de fichier et de manière générale tout ce qui peut avoir une valeur initiale. Ceci vous permet de tout mettre en place. Vous pouvez également initialiser les variables dans le bloc begin...end qui se trouve à la fin de l'unité.

La section finalization est l'opposée de la section précédente. Cette section vous permet de faire un peu le ménage avant de refermer l'application. Vous pouvez ainsi fermer des fichiers, désallouer de la mémoire et autres activités ménagères. Une fois exécutée la section initialization de votre unité, le code de votre section finalization s'exécutera à coup sûr avant la fermeture de l'application. Delphi 3 exécute les sections de finalisation des unités qui ont été utilisées dans l'ordre inverse de l'exécution des sections d'initialisation. Si vous initialisez les unités X, Y puis Z, elles se refermeront dans cet ordre : Z, Y puis X. Ceci est nécessaire si des unités contiennent d'autres unités dans leur déclaration `uses`. En effet, dans ce cas les unités ainsi dépendantes devront attendre que leurs sections de finalisation s'exécutent.

Info *Il est important que vous compreniez bien l'ordre dans lequel les différentes sections sont exécutées dans une unité. Lorsque votre application démarre, la section d'initialisation commence son exécution, dans l'ordre des noms d'unités qui figurent dans la déclaration Uses du programme principal. A partir de là, le code des unités est exécuté comme s'il était appelé par votre programme principal. Lorsque l'utilisateur ferme votre application, la section de finalisation de chaque unité est appelée, dans l'ordre inverse de celui des unités qui figuraient dans les sections d'initialisation.*

Voici un exemple d'unité permettant d'effectuer deux opérations mathématiques simples. Cette unité n'a pas beaucoup d'applications pratiques, mais elle illustre bien la structure et la fonction d'une unité.

```
Unit Maths;

interface
 function AjouterDeuxNombres (Un, Deux : Integer) : Integer;

 function SoustraireDeuxNombres (Un, Deux : Integer) : Integer;

function MultiplierDeuxNombres (Un, Deux : Integer) : Integer;

 procedure PositiveKarma;

implementation

 function AjouterDeuxNombres (Un, Deux : Integer) : Integer;

   begin
      AjouterDeuxNombres := Un + Deux
   end;
```

```
function SoustraireDeuxNombres (Un, Deux : Integer) : Integer;

   begin
      SoustraireDeuxNombres := Un - Deux
   end;

function MultiplierDeuxNombres (Un, Deux : Integer) : Integer;

    begin
       MultiplierDeuxNombres := Un * Deux
    end;

 procedure PositiveKarma;
    begin
       WriteLn('Tu peux le faire, les maths c''est de la rigolade.')
    end;

end. {de l'unité Maths}
```

Cette unité simple montre bien la forme que prend une unité. Vous avez défini les fonctions et procédures qui sont disponibles pour l'utilisateur de l'unité de la section d'interface. Dans la section implementation, vous créez les éléments que vous avez annoncés dans la section d'interface. J'ai placé là la procédure PositiveKarma en pensant à ceux d'entre vous qui haïssent les maths. Vous verrez que Delphi 3 fait un usage forcené des unités.

Pour appeler cette unité, il vous suffit de l'inclure dans la section Uses de votre programme principal. Le Listing 3.24 montre un exemple d'appel à notre unité Maths.

Listing 3.24 : Exemple de programme utilisant des unités

```
program ExempleMaths;

uses
 Maths;

var
 A, B : Integer;

begin
 A := 1;
 B := 2;
 WriteLn ('La somme de ', A, ' et ',  B, ' est ', AjouterDeuxNombres(A,B));
```

```
  ReadLn
end.
```

Ce programme utilise l'une des fonctions de `Maths`, `AjouterDeuxNombres`, afin de vous montrer comment utiliser une unité. Une fois l'unité ajoutée à votre projet, vous pouvez appeler toutes les fonctions qu'elle contient. Pour ajouter une unité à un projet, sélectionnez Fichier | Utiliser Unité, ou bien passer par le Gestionnaire de projet et cliquer sur le bouton Ajouter. Vous en apprendrez plus à ce sujet au Jour 5.

Réutilisation

Les concepts de réutilisation du logiciel et de bibliothèques de composants ont émergé ces dernières années. L'unité est une extension naturelle de cette théorie de la réutilisation. En effet, une unité permet au développeur de créer un ensemble de routines générales qu'il peut mettre de côté pour l'utiliser à sa guise par la suite.

La déclaration Uses vous permet d'inclure vos propres unités dans votre application. Delphi 3 propose un ensemble d'unités standard qui se chargent de fonctions générales, telles que les E/S de fichiers, les formes, les graphismes, les boutons, et bien d'autres encore (une liste complète des unités proposées par Delphi figure dans l'aide en ligne). L'usage d'unités procure plusieurs avantages. Comme en grande partie les fonctionnalités d'une application peuvent être divisées en plusieurs groupes ou zones, il semble logique d'adopter un modèle de programmation qui suive ce concept.

Les unités rendent également plus facile la phase de débogage. Si vous rencontrez une difficulté avec votre formule de maths, il vous suffit de consulter votre unité mathématique pour déboguer la fonction, au lieu de devoir fouiller dans la totalité de votre application pour dénicher l'erreur. La capacité à fragmenter votre programme vous permet de regrouper fonctions et procédures dans des unités et ainsi de mieux organiser votre projet. Dans un projet de taille importante et où de nombreuses personnes sont appliquées, vous pouvez même désigner un bibliothécaire de code chargé de conserver les dernières versions de vos unités et de les distribuer.

Distribution et sécurité

Comme l'écriture d'un livre, l'écriture d'un logiciel consiste à créer quelque chose en ne partant de rien (ou presque). Cet effort de création doit être protégé. Si vous découvrez un algorithme de chiffrage révolutionnaire, vous ressentirez vite le besoin de protéger votre code, tout en ayant la possibilité de le vendre. Vous voilà face à un dilemme : comment vendre votre code à d'autres développeurs sans pour autant leur fournir le code source et risquer ainsi de révéler vos algorithmes ?

Les unités sont un moyen parfait pour ceux qui désirent distribuer leur code sans pour autant l'exposer au piratage. Les unités Delphi peuvent être compilées en fichiers binaires et distribuées sous cette forme. Lorsqu'une unité est compilée, Delphi lui donne un suffixe .DCU. Cela indique qu'il s'agit d'une Unité Compilée Delphi (Delphi Compiled Unit en anglais). Vous pouvez distribuer votre unité sous cette forme, et d'autres personnes pourront utiliser votre unité pour leurs applications (en l'incluant dans la déclaration Uses), sans pour autant voir le code source. Ceci vous permet de développer votre code et de le commercialiser en toute sécurité.

Pour que des développeurs puissent utiliser votre unité, ils doivent connaître les fonctionnalités qu'elle propose. Il est donc nécessaire que vous décriviez en détail ces fonctionnalités dans un document accompagnant l'unité. De nombreux développeurs se contentent de copier la section interface de leur unité pour la distribuer comme documentation (en effet, puisque l'unité est compilée, la section interface n'est plus lisible). Il existe un véritable marché pour les unités, les DLL et les VCL et vous pourriez très bien y prendre pied un jour ou l'autre.

Info

Il convient de préciser toutefois que jusqu'à présent, ce concept de distribution des unités n'a pas toujours bien fonctionné lorsque les versions des produits changeaient. Il est généralement nécessaire de recompiler des unités pour chaque version de Pascal/Delphi. C'est pour cette raison que de nombreux programmeurs mettent à la disposition des acheteurs leur code source (moyennant finances bien sûr).

Développement en équipe

Un autre avantage de la fragmentation de votre projet en unités est la capacité d'affecter différentes tâches aux développeurs d'une équipe. Une fois l'application divisée en unités chargées des maths, du graphisme, des entrée/sortie et ainsi de suite, vous pouvez affecter un développeur à chaque unité. Le spécialiste des maths s'occupera de l'unité mathématique, le spécialiste des graphismes s'occupera de l'unité correspondante, et ainsi de suite. La plupart des applications deviennent trop grandes pour qu'il soit possible d'en confier le développement à une seule personne (étant donnés les délais de plus en plus resserrés). Il est indispensable de trouver des moyens de partager la charge de travail sur plusieurs développeurs et d'obtenir plus vite un produit, comme indiqué Figure 3.1.

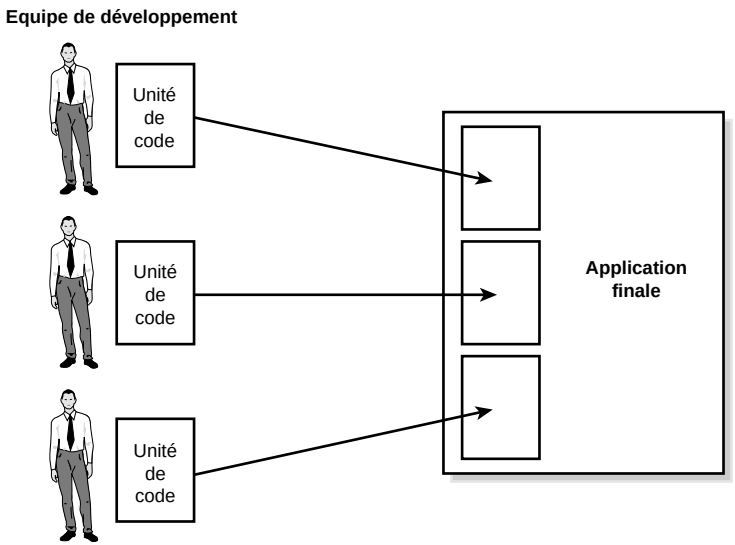


Figure 3.1
Le concept de développement en équipe.

Pointeurs

Ne vous culpabilisez pas si vous n'êtes pas devenu un expert des pointeurs à la fin de cette journée. Il faut de la pratique avant de bien les maîtriser.

Lorsque vous créez des structures de données en Delphi 3, de la mémoire leur est allouée. Cette mémoire est nécessaire pour contenir les données de votre structure. La plupart des structures de données (tels les enregistrements et les tableaux) peuvent rapidement devenir très grandes. C'est pour cette raison, entre autres, qu'il convient de n'allouer que la mémoire que vous comptez effectivement utiliser.

Une variable pointeur est capable de stocker l'adresse d'une structure de données (enregistrement ou tableau par exemple). Ceci revient à regarder quelque chose dans l'annuaire téléphonique. Vous pouvez donner ce pointeur vers vos données au lieu des les données elles-mêmes à d'autres fonctions et procédures. Quel en est l'intérêt ? Pensez à une carte. Vous n'apportez pas tout le contenu du magasin à un voisin pour lui conseiller le magasin, vous vous contentez de lui donner l'adresse du magasin, ou son "pointeur".

Les procédures font une grande utilisation des pointeurs. Vous utilisez des pointeurs chaque fois que vous faites un appel de fonction et utilisez le désignateur var dans la déclaration de procédure formelle. Lorsque vous passez des paramètres en utilisant var, ce désignateur passe en fait un pointeur vers les données de votre procédure. C'est la raison pour laquelle lorsque vous modifiez les données qui se trouvent dans une variable paramètre var dans votre procédure, les données sont modifiées également en dehors de la procédure. Vous modifiez les données extérieures directement, en utilisant ce "pointeur" vers la donnée.

Cette façon de faire a un grand avantage. Si vous n'utilisez pas le désignateur var dans la liste de paramètres formels de votre appel de procédure, chaque fois que vous passez quelque chose à la procédure, Delphi 3 doit effectuer une copie locale des données pour les manipuler. Si vous n'utilisez pas var, vous déclarez en fait "ne touche pas à mon original". Imaginez un peu la mémoire nécessaire à la copie d'un tableau contenant 10 000 enregistrements. Sans pointeurs, le prix à payer est bien lourd.

Info *Il y a aussi un prix à payer pour les pointeurs. Ces derniers utilisent aussi des ressources système. Pour référencer les données, Delphi doit à chaque fois dé-référencer un pointeur. Au mieux, cela peut prendre deux ou trois fois plus de temps CPU qu'il n'est nécessaire pour effectuer l'opération. Là encore, la sagesse veut que l'on n'utilise que ce dont on a besoin.*

Utiliser des pointeurs

Pour utiliser un pointeur, vous devez d'abord le définir. Prenons un exemple simple de pointeur vers un nombre réel, comme le montre le Listing 3.25.

Listing 3.25 : Exemple de programme utilisant un pointeur

```
program ExemplePointeur;

Uses
 Forms;

type
 PointeurReel = ^Real;

var
 P : PointeurReel;

begin
 New (P);
 P^ := 21.6;
 WriteLn (P^ :5:1);
  Dispose (P)
 ReadLn
end.
```

Vous voyez ici un exemple simple d'utilisation de pointeur. Vous commencez par créer un type appelé PointeurReel. Pointeurreel est un pointeur vers un nombre réel. Le chapeau ^ veut dire "pointant vers". Une fois le pointeur défini, vous devez créer une variable de ce type. Vous créez donc la variable P de type Pointeurreel. Vous disposez maintenant d'une variable qui est un pointeur vers un nombre réel.

Vous devez commencer par allouer de la mémoire à P. Pour l'instant, P est capable de pointer vers un nombre réel, mais il ne pointe encore sur rien. En utilisant l'appel de procédure New(), vous demandez à Delphi 3 d'affecter un bloc de mémoire capable de contenir un nombre Real, et de placer cette adresse mémoire dans P. P pointe maintenant vers un emplacement de mémoire contenant un nombre réel. La ligne P^ := 21.6 doit se lire "Définissez l'emplacement sur lequel pointe P comme étant égal à 21.6". Ceci s'appelle *déréférencer* un pointeur. Autrement dit, vous prenez la valeur 21.6 et la placez dans l'emplacement de mémoire sur lequel pointe P.

Lorsque vous utilisez le WriteLn pour afficher la valeur de ce sur quoi pointe P, vous devez encore une fois utiliser le caret (le chapeau) pour dire "Affichez ce sur quoi P pointe, et formatez-le sur 5 chiffres avec un chiffre à droite de la virgule". Une fois que P ne vous sert plus à

rien, vous devez utiliser l'appel de procédure Dispose() pour libérer la mémoire sur laquelle pointe P et pour la rendre au pool de mémoire disponible. Vous terminez alors votre programme.

Vous pouvez également utiliser des pointeurs pour pointer sur des objets plus complexes que de simples nombres réels. Dans le Listing 3.26, vous créez un enregistrement et utilisez un pointeur vers lui pour accéder à ses champs.

Listing 3.26 : Exemple de programme utilisant des pointeurs (2)

```
Program ExemplePointeur2;

Uses
 Forms;

type
 TypePersonne = RECORD
    Nom       : String;
    Prénom      : String;
    Age      : Integer
    end; {PersonRecord}

PointeurPersonne = ^TypePersonne;

var
 Person : PointeurPersonne;

begin
 New (Person);
 Person^.Nom := "Smith";
 Person^.Prenom := "James";
 Person^.Age := 35;
 WriteLn ('He bien, ', Person^.Prenom, ' ', Person^.Nom, ', vous avez ',
         Person^.Age, ' ans.');
  Dispose (Person);
ReadLn
end.
```

Ce programme ressemble beaucoup au précédent, la seule différence portant sur les types de données.

Vous avez créé un type appelé TypePersonne qui est un enregistrement comportant un nom, un prénom et un âge. Ensuite, vous créez un type PointeurPersonne qui est un pointeur sur l'enregistrement que vous venez de créer. La création de la variable Person vient ensuite. Cette variable est de type PointeurPersonne. Au début de l'exécution du programme principal, vous

passez par les mêmes phases que dans le programme ExemplePointeur. L'allocation de mémoire est assurée par la commande New(). Vous placez des valeurs dans l'enregistrement sur lequel pointe Person en utilisant le ^ pour déréférencer le pointeur et montrer dans quel champ de l'enregistrement placer les données. Une fois les valeurs affectées à l'enregistrement, le résultat est affiché dans une chaîne du type "James Smith, vous avez 35 ans". La commande Dispose() libère la mémoire allouée à Person.

En vérité, Windows est rempli de pointeurs. Le plus souvent, les programmes s'échangent des données au moyen de pointeurs. Dans des langages tels que C++ les pointeurs sont à la base de tout. Nous n'avons fait qu'effleurer le sujet et nous pourrions consacrer un livre tout entier aux pointeurs. Lisez tout ce que vous pouvez trouver concernant les pointeurs et regardez les exemples. Delphi a fait de gros efforts pour cacher les pointeurs, mais il viendra un temps où vous ne pourrez plus y couper, tant le gain de productivité peut devenir important dans certaines situations.

Récapitulatif

Vous devez maintenant connaître les concepts nécessaires au développement de programmes modulaires. Nous avons parlé du développement en équipe et des avantages qu'il y avait à utiliser des unités, des procédures et des fonctions.

Atelier

L'atelier vous donne trois façons de vérifier que vous avez correctement assimilé le contenu de cette leçon. La section Questions - Réponses présente des questions qu'on se pose couramment et leurs réponses, la section Questionnaire vous pose des questions, dont vous trouverez les réponses en consultant l'Annexe A, et vous mettrez en pratique dans les Exercices ce que vous venez d'apprendre. Tentez dans la mesure du possible de vous appliquer à chacune des trois sections avant de passer à la suite.

Questions - Réponses

Q Pourquoi le Pascal Objet propose-t-il des commandes telles que Goto si on considère que leur usage trahit une mauvaise méthode de programmation ?

R La polémique concernant le bon usage de Goto n'a pas encore pris fin (et elle ne prendra sans doute jamais fin). Avant que tous, d'un commun accord, décident de s'en débarrasser, l'outil reste là, utilisable par qui veut.

Q Qu'arriverait-il si une procédure renvoyait le contrôle à la ligne de code qui a appelé la procédure et non à la ligne qui la suit ?

R Cela ne fonctionnerait pas. A chaque fois que la procédure finirait, elle renverrait le contrôle à une ligne de code qui rappellerait la procédure.

Questionnaire

1. Combien de fois la déclaration WriteLn() sera-t-elle exécutée dans l'exemple ci-après ?

```
Program Questionnaire 4_1;

var
 Compte, Compte2 : Integer;

begin
 For Compte := 1 to 10 do
   For Compte2 := 1 to 10 do
      WriteLn ('Hello!');
end.
```

2. Quelle instruction de boucle utiliserez-vous si vous souhaitez tester votre clause conditionnelle avant d'entrer dans la boucle ?
3. Quelle est la différence fondamentale entre une procédure et une fonction ?
4. En quoi est-il plus avantageux d'utiliser des pointeurs dans une application plutôt que de transmettre les données elles-mêmes ?

Exercices

1. Ecrivez une application qui compte jusqu'à 42, affiche le nombre à mesure qu'il compte et utilise deux boucles For pour cela.
2. Essayez d'écrire une application comportant plusieurs unités. Pour vous habituer à la déclaration Uses, faites que votre application principale utilise une unité et que cette unité utilise elle-même une autre unité. Du moment qu'elles sont compilées, ces unités peuvent être aussi simples que vous le voulez.
3. Ecrivez une procédure qui prenne plusieurs paramètres, utilisant le passage par référence et par valeur. Faites des expériences pour voir comment la modification de l'ordre de transmission des paramètres peut créer différents messages d'erreur de compilation.

4

La programmation orientée objet

Nombreux sont ceux qui ont acheté ce livre et découvert Delphi (ou le Pascal Objet) et qui, d'autre part, ont peu d'expérience en matière de développement de logiciel, et à plus forte raison, de génie logiciel. Ce n'est pas sans importance car pour apprendre à être un bon informaticien, il ne suffit pas d'apprendre la syntaxe d'un langage particulier, tel que Delphi.

Le génie logiciel ne se résume pas à la simple écriture du code mais comporte d'autres facettes tout aussi importantes. Lors de cette journée, nous aborderons plusieurs notions qui peuvent ou non vous être familières, mais dont l'importance n'est pas à négliger. Ces notions comprennent la crise du logiciel, le cycle de vie du logiciel et le génie logiciel.

Vous avez peut-être envie de me répondre que tout ce qui vous importe, c'est d'apprendre Delphi. Peut-être bien, mais si vous souhaitez un jour gagner de l'argent en écrivant vos propres logiciels, ou même seulement faire partager vos créations, vos clients attendront de vous que :

- Votre logiciel fonctionne immédiatement.
- Votre logiciel soit fait à temps.
- Votre logiciel soit sans bogues.
- Votre logiciel soit bon marché.

Autant de raisons pour lire cette partie. Evoquons ensemble les problèmes qui se posent aujourd'hui, les solutions potentielles à ces problèmes, et de quelle manière Delphi 3 peut mettre en œuvre ces solutions.

La crise du logiciel

La crise du logiciel dure depuis si longtemps que l'on a fini par penser qu'elle représentait un état normal. On ne programme des ordinateurs que depuis quelques décennies. Pourtant, et tout amateur d'informatique vous le dira, en deux ou trois ans la technologie fait des bonds extraordinaires. Les temps changent, la technologie change, mais pas les gens. Les méthodes de programmation ont très peu évolué depuis les débuts de l'informatique. Cependant, assez récemment, on a pu assister à un bouleversement brutal de la doctrine dominante en matière de programmation.

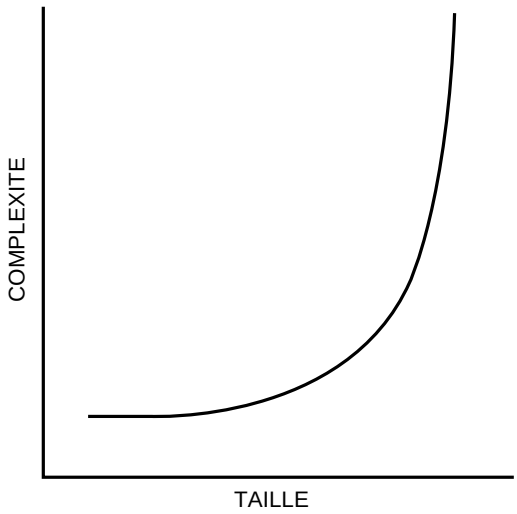
Autrefois (avant 1990), lorsqu'un programmeur avait une tâche à effectuer, il se plantait devant son clavier et commençait à coder. C'était le bon vieux temps. En ce temps-là, si vous aviez quelque chose à faire, il vous suffisait de passer une ou deux nuits blanches devant votre clavier et le résultat était là. C'était l'époque des programmes de 1000 lignes. C'était l'époque où un simple individu pouvait pianoter un peu plus intelligemment que les autres et devenir millionnaire. Bref, c'était l'âge d'or de la programmation. Mais toutes les bonnes choses ont une fin.

Complexité du logiciel

Les ordinateurs sont devenus plus sophistiqués, et les utilisateurs aussi. La taille des tâches confiées n'a cessé d'augmenter, comme a augmenté par conséquent le temps de codage. Puis,

on s'est aperçu de quelque chose d'étrange. Passée une certaine taille de programme, le temps de codage s'avérait beaucoup plus long que prévu (en se basant sur des programmes plus courts). Si un programme était deux fois plus long qu'un autre, on supposait qu'il fallait un temps de codage double (on croyait que la fonction était linéaire). En fait, cela prenait beaucoup, beaucoup plus de temps (comme illustré Figure 4.1). Signe de la profonde injustice de l'univers ou application de la loi de Murphy, on a dû se rendre à l'évidence : lorsqu'un programme atteint une certaine taille, le cerveau humain ne peut embrasser toute la complexité de l'application. La complexité était donc en cause et l'on n'y pouvait rien (rien directement, tout du moins). Cette triste constatation s'imposa vite.

Figure 4.1
Taille d'un logiciel et complexité associée.



Vous avez pu remarquer que les boutiques regorgent de logiciels très complexes. Est-ce que cela veut dire que l'on a résolu ce problème de complexité ? Oui et non... La complexité est semblable à l'océan, indomptable et imbattable. Vous devez naviguer dans la complexité, profiter des périodes de calme et chercher abri dans un havre sûr lorsque le temps se gâte. A votre avis, combien de ces fabuleux logiciels effroyablement complexes sont totalement exempts de bogues ? Très peu. L'existence du bogue vous rappelle que la complexité n'a pas été battue. On trouve même des bogues dans les programmes les plus simples. Il faut un effort non négligeable pour débarrasser le plus petit des programmes de tous ses bogues. Imaginez un peu ce que ça doit être dans un programme complexe. En fait, la solution consiste à écrire des petits programmes "sans bogues" et de les regrouper pour former un gros programme "sans bogues". Plus facile à dire qu'à faire, bien sûr, mais cette approche modulaire nous permet de maîtriser la complexité. Vous avez remarqué les guillemets autour de "sans bogues". Un logiciel "sans boguess" est un peu comme de l'or "100 % pur" : ça n'existe pas.

Planifier

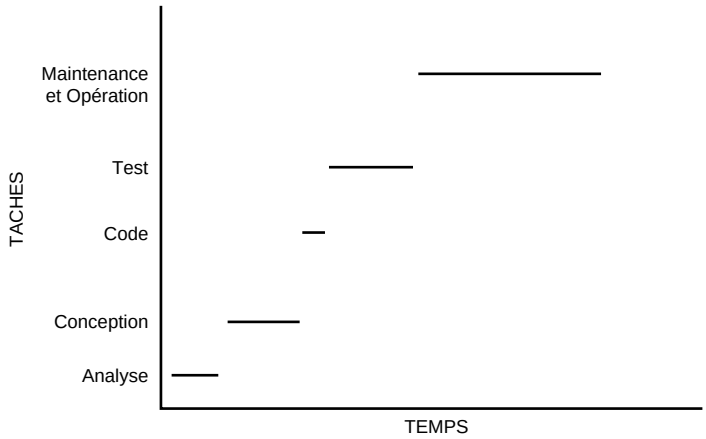
Il est clair que nous avons fini par nous rendre compte qu'il ne suffisait pas de s'asseoir et de se mettre à coder. Il est devenu temps d'appliquer l'analyse scientifique à ce problème, et de ne plus s'en remettre à l'inspiration. Le cycle de vie du logiciel et le génie logiciel nous ont permis de progresser.

Cycle de vie du logiciel

Le cycle de vie du logiciel correspond à une série d'étapes à suivre, et à suivre dans l'ordre. Cette approche disciplinée permet au développeur de créer un meilleur produit, mais aussi de passer moins de temps à assurer l'entretien de son produit. Ce n'est pas une panacée, mais un outil puissant.

Le cycle de vie du logiciel est séparé en six domaines principaux. Ces domaines sont repris Figure 4.2. En les lisant, vous verrez qu'il ne s'agit que de bon sens. Mais vous seriez surpris de voir le nombre de développeurs qui sautent plusieurs étapes et commencent à coder immédiatement, puis sautent également les étapes suivant le codage. Examinez chaque étape et essayez de comprendre ses implications.

Figure 4.2
Le cycle de vie du logiciel.



Analyse

Lors de la phase d'analyse du développement, vous devez vous efforcer de mieux définir le problème à résoudre et décider si après tout une solution logicielle est bien adaptée. De nombreuses choses peuvent être résolues bien plus facilement avec une feuille de papier et un crayon qu'avec un logiciel.

Vous avez certainement vu des applications vous demandant une masse de travail bien trop importante pour les résultats obtenus. Il faut s'assurer qu'une solution informatique résout le problème au lieu d'y ajouter.

Lors de cette phase d'analyse, vous devez également déterminer le type de ressources dont vous aurez besoin pour mener le projet à terme. Il se peut que seuls vous, votre PC et Delphi2 soient nécessaires. Pour des projets plus importants, vous aurez peut-être besoin de l'assistance d'autres développeurs, de rédacteurs techniques, etc. Il n'est pas encore question de choisir un langage. En effet, on ne peut choisir un langage qu'une fois le problème bien cerné et la solution conçue. Après quoi, on peut alors sélectionner le langage le mieux adapté à la solution envisagée.

Sur de gros projets, le résultat de cette phase d'analyse débouche généralement sur la création de spécifications système. Ces spécifications permettent de définir le système dans son ensemble, en établissant quelles seront les fonctions système qui devront être effectuées.

Lors de cette phase, on ajoute souvent quelques détails spécifiques demandés par l'utilisateur. Ces détails sont pris en compte afin de lui éviter de se fourvoyer dans une impasse de développement, et sont rarement liés à l'implémentation en elle-même. Ils concernent plutôt des remarques générales d'ergonomie dont on déduit l'approche correcte à utiliser pour l'intégralité de l'analyse.

Cahier des charges

Dans cette phase, vous prenez les fonctions définies dans la phase d'analyse et déterminez un cahier des charges détaillé pour chacune d'entre elles. Les spécifications système servent de base pour cette phase, et des spécifications logicielles précises en sont le résultat. Quelles fonctions le logiciel doit-il accomplir ? Ici, vous ne concevez pas la solution complète, uniquement les fonctions du logiciel.

A ce stade, vous pouvez commencer à penser aux fonctions en termes de déclarations d'unités, de fonctions et de procédures Delphi. Si vous avez déjà choisi votre langage, en utilisant ici la syntaxe de Delphi, vous créerez une documentation mieux intégrée à la solution et plus facile à modifier.

Conception

C'est dans cette phase que tous les détails de la conception sont affinés. Comme vos idées sont claires, suite à la phase du cahier des charges, vous pouvez passer à l'étape suivante. Elle consiste à développer un design détaillé de votre solution logicielle, comprenant une définition des relations entre les diverses unités et un flux procédural détaillé.

A mesure que vous progressez, vous devez définir et décrire la façon dont tout fonctionnera. Vous devez mettre au point des exemples d'écran, la disposition des boutons, etc.

Il est important que vous compreniez que votre client ne connaîtra pas tous les détails de prime abord (un client sait rarement ce qu'il désire, il sait simplement qu'il désire). C'est un processus itératif. Il est important de communiquer avec les destinataires du logiciel, pour se mettre d'accord sur le projet.

Un des avantages à utiliser Delphi comme langage de développement est que vous pouvez l'utiliser lors de ces phases comme une sorte de langage de définition permettant de montrer la logique générale de votre application. Si vous êtes un tant soit peut anglophone, vous pouvez presque comprendre ce qui se passe en lisant simplement le code.

Codage

Cette phase de développement commence ici et ne se termine jamais... Vous continuerez à coder pendant tout le cycle de vie du logiciel.

Si vous avez bien conçu votre logiciel et correctement défini les interfaces des différents modules, le code devrait s'écrire tout seul ou presque.

Cette phase aboutit à deux résultats. Tout d'abord, le code lui-même bien sûr, mais également, presque aussi important, des spécifications complètes du logiciel décrivant le produit tel qu'il a été réalisé. Ceci servira d'outil de test et servira de base pour le manuel de l'utilisateur.

Test

Le but de cette phase est de tester le produit tel qu'il a été créé, et de vérifier sa conformité au cahier des charges. C'est lors de cette phase que vous savez si vous avez réussi à faire de vos désirs une réalité. Chaque fonction de l'application doit renvoyer à une spécification du cahier des charges. Si une fonction ne correspond à aucune spécification, cette fonction n'a pas de raison d'être.

On effectue généralement les tests en deux temps. Tout d'abord on procède au *test d'unités*. Cette phase consiste à prendre séparément chaque unité, à écrire des pilotes pour chacune et à tester leur fonctionnalité et leur fiabilité. Ensuite vient le *test d'intégration*. Dans cette phase, vous réunissez les unités déjà testées et vous vérifiez qu'elles se comportent ensemble conformément à vos attentes.

Le test des logiciels est un art en soi. Si vous écrivez une application simple, les tests pourront être tout aussi simples. Dans le cas de programmes plus importants, les tests peuvent devenir très lourds à mener à bien si vous les menez sans discernement. Il existe de nombreux livres consacrés exclusivement aux tests de logiciel.

Maintenance et opération

Vous pensez peut-être en avoir fini. Le produit a été livré et vous êtes soudain débarrassé d'un sacré poids. Pas si vite... Vous devez savoir que 30 % des efforts et de l'argent sont dépensés dans cette nouvelle phase. Vous devez maintenant suivre le parcours de votre logiciel chez les clients, qui trouveront des fonctions non documentées (un euphémisme pour bogues) auxquelles vous devrez remédier. Puis votre application prendra doucement le chemin de l'obsolescence. Il est important de garder à l'esprit que plus le cycle de vie d'un produit raccourcit, plus les coûts de maintenance baissent. Il vous faudra par contre vous adapter à cette nouvelle vitesse d'évolution, qui impose l'ajout constant de nouvelles fonctionnalités à vos logiciels, ainsi que la rédaction et la maintenance des documentations liées au produit.

Cela n'a rien de très gai effectivement car il n'est pas aussi amusant de maintenir un logiciel que de l'écrire, mais c'est indispensable. Comment le nouveau qui vient d'arriver pourra-t-il assurer la maintenance de votre application une fois que vous serez attelé à un nouveau projet ?

Toute la documentation, manuel de l'utilisateur, documents de conception, cahier des charges et rapports de tests aideront le nouveau venu dans sa tâche. Faire correctement votre travail signifie aussi vous arranger pour que vous puissiez passer à autre chose.

Et ensuite ?

Maintenant que nous avons évoqué certaines des étapes nécessaires à une approche méthodique, nous devons parler du niveau supérieur. Ce qui était nécessaire autrefois (et qui le reste aujourd'hui selon certains) était une discipline de programmation, c'est-à-dire un ensemble de règles servant de guide au programmeur, et aidant à suivre le processus de conception et à l'affiner sans cesse avec de nouvelles règles permettant d'obtenir un environnement plus stable.

Le génie logiciel

Un ingénieur logiciel est très différent d'un programmeur. Un ingénieur logiciel discute avec le futur utilisateur et écrit noir sur blanc les besoins de l'utilisateur. Il écrit ensuite noir sur blanc ce qui doit être fait pour construire le logiciel. Après tout cela seulement l'ingénieur logiciel commence à écrire son code.

Ce n'est peut-être pas toujours passionnant et nécessite souvent beaucoup de travail, mais cela permet souvent d'écrire un code très solide. Il est bien sûr beaucoup plus amusant de s'asseoir devant son ordinateur et de se mettre à bricoler du code, mais dans les gros systèmes il est encore plus amusant d'avoir, au bout du compte, un logiciel qui fonctionne. Le génie logiciel est la version disciplinée de la programmation.

Objectifs du génie logiciel

Si vous vous sentez prêt à adopter cette démarche, vous devez vous fixer un but. Si vous ne savez pas où vous allez, vous aurez du mal à faire quoi que ce soit. Le premier objectif est que le produit fini corresponde au cahier des charges. Mais c'est un but trop général que vous devez subdiviser en plusieurs objectifs plus limités, mais aussi plus faciles à réaliser et à garder à l'esprit.

N'oubliez pas que les choses changent constamment. Comme nous l'avons dit auparavant, votre application passera la majeure partie de son cycle de vie dans la phase de maintenance. Il est donc important que notre ensemble d'objectifs prenne en compte les changements. Les quatre objectifs du génie logiciel sont donc la capacité de modification, l'efficacité, la fiabilité et la compréhensibilité.

Capacité de modification

Dans le cycle de vie de votre logiciel, celui-ci devra être modifié à de nombreuses reprises. Ce peut être à la suite de la découverte d'un bogue par vous ou par un client, ou pour ajouter une fonction demandée après coup par les utilisateurs. Dans tous les cas de figure, il est important que votre travail, qui résulte d'un processus de conception rigoureux, ne soit pas détruit si vous apportez la moindre modification.

Vous devez pouvoir apporter des modifications à votre application sans endommager les fondations. Il faut donc contrôler les modifications, en changeant certaines parties de votre application sans altérer les autres. Ce n'est pas chose facile. De nombreux langages ne permettent pas de le faire correctement et réagissent très mal à la moindre modification.

Le Pascal Objet est quant à lui un langage très lisible. Il fournit également un environnement vous permettant de procéder facilement à des modifications. Les typages de chaînes minimisent les risques d'erreurs. Tous ces facteurs font que cet environnement est parfaitement adapté à la capacité de modification. Lorsque vous effectuez une modification vous devez toujours penser à la répercuter dans votre documentation. Vous pouvez même pousser le zèle jusqu'à ajouter la nouvelle spécification correspondante au cahier des charges, puis suivre cette spécification dans votre document de conception, avant d'effectuer la modification dans le code lui-même. Cette façon de faire, aussi fastidieuse soit-elle, augmentera la stabilité de votre application.

Efficacité

Ce principe est simple. Votre application doit utiliser de façon optimale les ressources disponibles. A l'époque des ordinateurs disposant de 64 ko, la mémoire dont nous disposions était très limitée mais comme les systèmes d'exploitation n'étaient pas multitâche, ce peu de mémoire vous appartenait entièrement.

Aujourd'hui, la plupart des applications doivent s'exécuter simultanément. C'est pourquoi il est important que votre application fasse attention aux ressources et interagisse correctement avec d'autres applications. Le termes *ressources* s'applique à la fois à des ressources temporelles et spatiales. Votre application peut avoir à exécuter certaines choses en un temps requis, si elle collecte des données évoluant rapidement ou doit reprendre des dépêches d'agence. De plus, vous êtes aussi limité par le temps puisque vous devez laissez des cycles CPU à d'autres. Votre application doit savoir partager ce temps avec d'autres. Si elle thésaurise les cycles CPU, regardez l'aide concernant la fonction `Application.ProcessMessages()`.

Deuxième ressource précieuse : l'espace. Comme vous avez dû le remarquer, les logiciels deviennent de plus en plus gros. La plupart d'entre eux sont maintenant fournis sur CD-ROM car il serait trop coûteux d'expédier des lots de quarante disquettes. Là encore, tâchez de vous modérer et de ne prendre que ce dont vous avez besoin.

Fiabilité

C'est probablement l'objectif le plus important, tout particulièrement si votre application se charge de tâches importantes au sein d'une entreprise. Les applications qui s'exécutent pendant

de longues périodes de temps sans intervention humaine doivent être stables et même pouvoir récupérer automatiquement en cas de problème.

Le coût des erreurs est tel qu'il vous faut un logiciel aussi fiable que possible. Vous devez dès le début de la conception assurer la fiabilité de l'ensemble. Il existe de nombreux moyens pour vous assurer de ne pas provoquer des messages de type `depassement numerique` ou `enregistrement invalide`. L'aspect fortement typé du Pascal Objet vous évite de nombreuses erreurs plus faciles à faire sous des langages plus "laxistes". Par ailleurs, en activant la vérification d'intervalle, vous pouvez éviter d'autres problèmes. En effet la vérification d'intervalle vous permet de vous assurer que les valeurs que vous pensez envoyer à vos variables sont vraiment envoyées. Ceci permet lors de la phase de test de dénicher bien des bogues.

Compréhensibilité

Cet objectif vient du monde la programmation orientée objet. Pour que votre application soit compréhensible, et donc modifiable, elle doit être simplifiée. Or, il n'est pas toujours évident de simplifier tous les mécanismes d'un système complexe.

Une façon de remplir cet objectif consiste à reproduire autant que possible la réalité dans les modèles de conception et d'implémentation de votre application. Si les objets de votre application renvoient à des objets de la vie réelle, vous pouvez plus facilement comprendre les relations entre vos objets virtuels. C'est la raison pour laquelle la programmation orientée objet est préférable aux techniques plus anciennes.

La lisibilité est une autre façon de rendre votre code compréhensible. L'usage de bonnes méthodes de programmation et de commentaires ajoute à la compréhensibilité. Les objets du code doivent être facilement repérables.

Principes du génie logiciel

Les objectifs dont nous venons de parler sont applicables à n'importe quelle application. Mais comment remplir ces objectifs ? Vous devez définir un ensemble de principes (repris dans le Tableau 4.1) sur lesquels vous vous appuierez pour atteindre vos objectifs.

Les sept principes du génie logiciel :

1. Modularité.
2. Localisation.
3. Abstraction.
4. Dissimulation de l'information.
5. Confirmabilité.
6. Uniformité.
7. Complétude.

Ces principes sont pris en charge dans la structure de classe de Delphi. Certains découlent naturellement des classes, tandis que d'autres doivent être suivis consciemment par le développeur.

Modularité et localisation

La modularité et la localisation découlent directement de la structure de classe. Les classes sont par essence modulaires, et la localisation suppose que les modules soient organisés logiquement, chaque module devant regrouper des fonctions proches. Si vous construisez votre code en écrivant des petits modules autonomes, ceux-ci pourront facilement être utilisés dans d'autres projets. C'est le principe du Référentiel d'objets.

Le Référentiel d'objets contient des objets qui peuvent être utilisés et réutilisés à l'envi. En faisant Fichier/Nouveau dans Delphi, vous avez accès au travail effectué pour vous par de nombreux autres programmeurs. La clé du succès ici réside dans la modularité et l'indépendance du code.

Abstraction et dissimulation de l'information

L'abstraction et la dissimulation de l'information sont très bien prises en charge par les classes Delphi. Les mots clé private et protected permettent deux niveaux de dissimulation des informations concernant des objets en dehors d'une classe donnée. L'abstraction est mise en place par le biais de la dissimulation de l'information et par la nature interface/implémentation des fichiers de classe. Vous ne laissez publique que la nature abstraite de la classe. Les détails de l'implémentation restent cachés.

Un exemple de ce principe est illustré par la façon dont vous travaillez avec des fichiers de texte en Delphi. Lorsque vous ouvrez un fichier, vous spécifiez le fichier à ouvrir et le mode (le mode indique que vous ouvrez le fichier pour le lire ou pour y écrire). Delphi ne vous demande pas que vous sachiez dans quel secteur du disque dur réside le fichier. Les détails d'implémentation du système de fichier vous sont cachés. Ceci vous permet de lire indifféremment des données provenant d'un disque dur, d'une disquette ou d'un CD-ROM, même si la structure physique de chacun de ces supports est différente des autres.

Confirmabilité

La confirmabilité est obtenue en combinant des vérifications de type poussées et la construction de modules testables. La vérification des types permet au compilateur de confirmer qu'une variable est utilisée correctement. Le typage des chaînes fait partie de Delphi. Les modules qui peuvent être testés permettent au programmeur de tester logiquement chaque module individuel. Souvenez-vous qu'il est bien plus facile de construire des petits programmes sans erreur. Tous les modules de codes doivent être présentés au Référentiel de code, accompagnés de programmes pilotes testant le module. La confirmabilité implique que vous puissiez subdiviser votre application en plusieurs modules testables.

Uniformité et complétude

L'uniformité et la complétude sont du ressort du programmeur. Un code est facile à lire et à entretenir s'il est écrit et commenté d'une manière uniforme. Les modules qui sont complets dès leur première rédaction n'ont pas besoin d'être réécrits ou complétés lorsqu'un nouveau besoin se fait sentir. La complétude, tout comme un code sans erreur, est très dure à obtenir. Mais si vous tendez vers cet objectif, vous pouvez rendre plus adaptable votre code et minimiser la nécessité d'y apporter des modifications par la suite. Les modifications sont le meilleur moyen d'ajouter des bogues dans votre application.

Tous ces principes du Génie logiciel ont pour but de combattre la complexité. Lorsque vous construisez vos classes, gardez ces principes à l'esprit, vous et votre code auront tout à y gagner. Et pensez également à ces principes quand vous examinez les classes de Delphi. Il arrivera souvent que vous vous demandiez pourquoi Borland s'est comporté si étrangement dans l'une de ses classes. Vous pouvez généralement trouver la réponse dans les principes décrits plus haut. La bibliothèque de classes de Delphi est un bon exemple de génie logiciel.

Accouplement et cohésion

Un des avantages de la méthode de programmation modulaire est que vous pouvez extraire un module de code d'une application pour le placer dans une autre. Supposez que vous ayez écrit une unité Delphi effectuant du chiffrement de données. Vous lui donnez un nom de fichier et le module lit ce fichier, le chiffre à l'aide d'une clé, et réécrit le fichier sur le disque.

En écrivant cette unité de code sous forme d'entité indépendante, vous pourrez l'utiliser dans de nombreuses applications en vous contentant d'appeler les fonctions de l'unité et de traiter les résultats.

Le terme de cohésion signifie que les fonctions internes de vos modules de code doivent être très intégrées. La deuxième notion est celle d'accouplement. Vos modules doivent être très peu accouplés à leurs voisins (en d'autres termes, ils doivent être très peu dépendants les uns des autres, voire pas du tout). Si votre unité dépend trop d'autres unités, elle ne pourra pas être portée vers une autre application sans devoir emporter de nombreux bagages avec elle. Votre code doit donc être très cohérent et très peu dépendant. Ce n'est pas impossible, mais demande du travail.

Conception orientée objet

Si vous regardez un peu le monde réel, vous verrez que notre langage est formé de deux composants principaux : les noms (objets) et les verbes (opérations). Pour que nos applications collent au plus près à la réalité, notre langage informatique doit adopter cette configuration. La plupart des langages comportent de nombreuses opérations, mais disposent de très peu de

"noms" pour décrire les objets. Et même si certains langages sont capables de traiter des objets, ces objets sont plats, c'est-à-dire incapables d'hériter des attributs de leurs ancêtres. Cela ne constitue pas une bonne modélisation du monde réel. Dans le monde réel les choses sont en relief, et nous avons donc besoin d'un langage qui reflète cet aspect des choses. Delphi offre un ensemble très complet de noms permettant de décrire des objets.

Comment définir un objet ? Ce mot est galvaudé. Selon Grady Booch (le maître de l'Orienté objet), un objet est : "une entité qui a un état, c'est-à-dire une valeur... le comportement d'un objet est défini par les actions qui l'affectent et vice versa... chaque objet est en fait une instance d'une classe d'objets". C'est un début.

L'objectif de la conception objet est que chaque module du système représente un objet ou une classe d'objets du monde réel. Grady Booch ajoute : "Un programme qui modélise la réalité peut donc être considéré comme un ensemble d'objets qui interagissent les uns avec les autres". Vous pouvez concevoir un système qui reprend cette approche orientée objet en suivant ces étapes :

1. Identifiez les objets et leurs attributs.
2. Identifiez les opérations qui affectent chaque objet, et les opérations que chaque objet doit initier.
3. Etablissez la visibilité de chaque objet par rapport aux autres.
4. Etablissez l'interface de chaque objet.
5. Implémentez chaque objet.

Objets

Lorsque vous identifiez les objets, vous devez penser en termes de noms. Dans un système de contrôle de chauffage, vous avez une source de chaleur, un détecteur de température, un thermostat et d'autres éléments. Ces noms deviennent les éléments principaux de votre système. Les objets peuvent être très grands et constitués d'objets plus petits. Ainsi, une voiture est un grand objet qui comprend des objets plus petits, comme le moteur, la transmission et la carrosserie.

Opérations

Vous devez identifier les opérations qu'effectue ou que subit chacun des objets définis lors de la phase précédente. Ainsi, un thermostat peut être réglé, un solénoïde peut être activé et un détecteur de température peut effectuer une mesure de température. Vous devez également définir quelles seront les opérations qui affecteront un objet en premier. Une automobile bien conçue démarrera toute seule en passant du point mort à la première.

Visibilité

Ici, vous définissez la topologie. Vous avez besoin de tracer une carte indiquant quels objets sont vus par quels autres. Dans notre exemple de chauffage, le détecteur de température doit

pouvoir être vu par le thermostat, mais le détecteur de température n'a pas besoin de voir le thermostat.

Interfaces

Ici, vous définissez la façon dont vos objets s'interfaceront avec les autres. Cette étape est très importante pour concevoir un système réellement modulaire. Vous devez définir avec précision la manière dont les autres objets communiqueront avec votre objet. Vous pouvez utiliser des déclarations d'appel de fonction ou de procédure pour définir l'interface de votre objet. Vous pouvez faire ceci si, et seulement si, le langage que vous utilisez est facilement lisible (c'est le cas pour Delphi).

Implémenter les objets

Dans cette étape, vous implémentez chaque objet dans votre solution. Vous devez donc écrire le code des interfaces de chaque objet. Vous n'êtes pas forcé d'écrire le code de l'objet tout de suite. Si votre objet est complexe (s'il est composé d'objets plus petits), vous devez décomposer l'objets en ces divers constituants. Pour chacun de ces objets constituants, vous devez reprendre les étapes précédentes pour déterminer leurs opérations, leur visibilité et leurs interfaces. Une fois que vous avez créé des squelettes fonctionnels pourvus d'interfaces bien définies, le codage du corps lui-même est une formalité.

En suivant ces étapes, vous pouvez concevoir un système bien pensé et à la cohésion assurée. Comme le Pascal Objet vous permet de coder en suivant cette méthode, tout est pour le mieux.

Programmation orientée objet (POO)

La programmation orientée objet existe depuis des années et a pointé le bout de son nez dans des langages tels que Ada, SmallTalk, C++, Borland Pascal et dans Delphi. Le terme "d'objet" a même acquis une connotation magique qui fait tomber en pâmoison le plus imperturbable des responsables informatiques.

La réalité est plus terre à terre. On peut trouver des codes orientés objet plus exécrables que le pire des codes écrits en Pascal standard. Il existe aussi des codes orientés objet de toute beauté. La programmation orientée objet n'est qu'un outil.

Plusieurs composants du Pascal Objet le rendent orienté objet. Examinons ensemble la définition de base des classes et quelques autres petits plus qui donnent au Pascal Objet son nom.

Classes

Delphi comporte un mot réservé, *class*, qui vous permet de définir un objet. Lorsque vous créez un nouveau projet dans Delphi, vous pouvez trouver dans les déclarations de `unit1` une déclaration de classe pour la fiche elle-même :

```
type
 TForm1 = class(TForm)
public
    { Déclarations publiques }
protected
    { Déclarations protégées }
private
    { Déclarations privées }
  end;
```

C'est ainsi que vous définissez un objet. Dans la section d'interface, utilisez votre nom de type (`TForm1`) puis la classe de base dont il dérive. Tous les objets doivent dériver de `TObject` ou de l'un de ses objets enfant.

La partie publique est réservée aux déclarations accessibles au monde extérieure. Dans la partie privée, vous pouvez déclarer des variables, des procédures et des fonctions qui ne seront utilisées que dans votre classe. La partie protégée vous donne tous les avantages des deux parties précédentes. Les composants qui sont déclarés protégés ne sont accessibles qu'aux descendants du type déclarant.

Comme dans le cas des composants privés, vous pouvez cacher les détails d'implémentation à l'utilisateur. Cependant, à la différence des composants privés, les composants protégés restent disponibles aux programmeurs qui souhaitent dériver de nouveaux objets à partir de vos objets sans être pour autant obligés de déclarer les objets dérivés dans cette même unité.

Créez votre propre objet de données. Stockez le code de votre carte bancaire dans l'objet. La déclaration sera :

```
Secret = class(TObject)
  private
    FCode : Integer;
end;
```

Vous pouvez remarquer que vous n'avez pas seulement déclaré une classe appelée `Secret`, mais que vous avez également une variable de données privées contenant la valeur du code lui-même. Nous avons appelé cette variable `FCode`.

Propriétés

Le mot réservé `property` vous permet de déclarer des propriétés. Une définition de propriété dans une classe déclare un attribut, pourvu d'un nom, pour les objets de la classe et les actions associées à la lecture et à l'écriture de l'attribut. L'attribut que vous avez créé est `FCode`. Vous ne souhaitez pas permettre au programmeur de modifier directement cette valeur. Il faut pour cela effectuer une entrée et valider cette entrée pour s'assurer que la modification est correcte. On retrouve là le principe de dissimulation de l'information développé plus haut.

Utilisez une propriété appelée `Code` qui servira d'intermédiaire lors des accès à votre valeur. Lorsque vous appelez la propriété `Code`, vous obtenez la valeur de `FCode`. La différence ici est que la variable `FCode` est protégée de toute modification extérieure. Il n'est possible de modifier sa valeur que par l'intermédiaire de la procédure `SetCode`.

```
Secret = class(TObject)
  private
    FCode : Integer;
  protected
     procedure SetCode (Nouveau_code : Integer);
     property Code : Integer read FCode write SetCode;
  end;
```

Vous pouvez maintenant demander la valeur du code sans le modifier. Les sélecteurs sont un élément nécessaire du modèle d'objet. Si vous ne pouvez pas voir les données d'un objet, il est sans utilité.

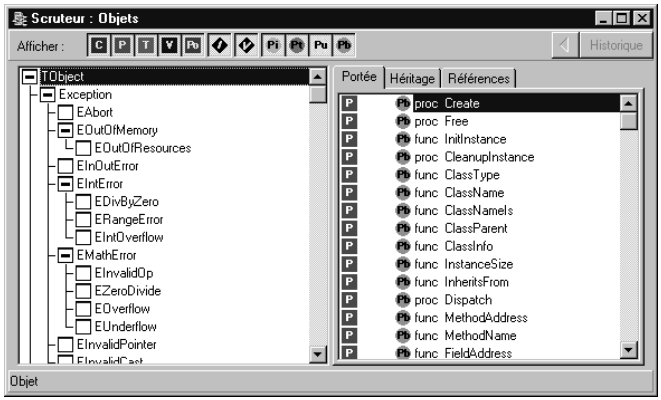
Héritage

Nouveau

L'héritage est une des fonctions les plus puissantes d'un langage orienté objet tel que Delphi. Il permet à des classes de descendants de bénéficier des propriétés de leurs ancêtres. Ils héritent des champs, des propriétés, des méthodes et des événements de leur classe ancêtre. En plus de disposer des attributs de ses ancêtres, la classe enfant peut leur ajouter ses propres éléments. Ceci vous permet de prendre une classe comportant la plupart des éléments fondamentaux dont vous avez besoin, et d'ajouter de nouveaux objets qui adaptent cette classe à vos besoins.

Si vous compilez un projet, même nouveau, vous pouvez ensuite aller dans Voir/Scruteur pour visualiser la totalité de la base d'objets de Delphi (voir Figure 4.3). Le volet Inspecteur (fenêtre gauche) vous permet de choisir une classe particulière de votre projet. Vous pouvez voir grâce aux arborescences quelles classes sont ancêtres des objets que vous avez sélectionnés. En utilisant l'onglet Héritage, vous pouvez voir tous les ancêtres et descendants de la classe sélectionnée dans le volet Inspecteur.

Figure 4.3
Le Scruteur d'objets



Récapitulatif

La programmation orientée objet n'est peut-être pas la panacée, mais elle peut nous rendre bien des services. La philosophie qui la sous-tend encourage l'organisation. Nous vous avons montré comment Borland a adopté l'Objet et vous permet de faire de même.

Lors de cette journée, nous avons évoqué les principes et les objectifs du génie logiciel. Vous avez pu voir le cycle de vie du logiciel et vous avez, sans doute, mieux compris les stades de l'évolution de votre projet. Enfin, nous avons découvert rapidement le monde des objets.

Cette leçon ne fait qu'effleurer des sujets qui mériteraient plus ample développement. Si cette partie a éveillé votre curiosité vous irez de vous-même à la recherche de plus de renseignements.

Je vous invite à vous intéresser également à d'autres techniques de développement logiciel. La technique de cycle de vie parallèle récursif, et d'autres, permettent d'aborder sous différents aspects les problèmes de développement logiciel. Utilisez la technique qui vous est la plus naturelle.

Lors du cinquième jour, vous apprendrez comment gérer une application Delphi du point de vue des fichiers : quels sont les fichiers qui composent une application Delphi, et comment utiliser l'environnement de Delphi 3 pour organiser vos projets.

Atelier

L'atelier vous donne trois façons de vérifier que vous avez correctement assimilé le contenu de cette leçon. La section Questions - Réponses vous donne des questions qu'on pose couram-

ment, ainsi que leur réponse, la section Questionnaire vous pose des questions, dont vous trouverez les réponses en consultant l'Annexe A, et les Exercices vous permettent de mettre en pratique ce que vous venez d'apprendre. Tentez dans la mesure du possible de vous appliquer à chacune des trois sections avant de passer à la suite.

Questions - Réponses

Q Peut-on appliquer les principes du génie logiciel avec un langage non orienté objet ?

R Oui, tout est possible. Mais si votre langage ne permet pas facilement d'établir une correspondance entre ses structures et le monde réel, il s'avérera difficile d'appliquer les principes cités plus haut. La maintenance du code posera elle aussi problème, car le code créé sera bien moins intuitif.

Q Peut-on faire de la programmation orientée objet sans faire de conception orientée objet ?

R Bien sûr, mais là encore c'est la cohérence de la démarche qui fait le succès d'un projet. Il vaut mieux choisir : tout ou rien (tout de préférence...).

Questionnaire

1. Quels sont les objectifs du Génie logiciel ?
2. De quelle manière le Pascal Objet assure-t-il la dissimulation de l'information ?
3. En quoi le principe d'héritage aide le Pascal Objet à modéliser l'environnement Windows ?

Exercice

1. Créez votre propre objet appelé `Personne` et contenant des informations concernant une personne (nom, adresse, numéro de téléphone par exemple). Créez ensuite une propriété qui récupère les valeurs de `Personne` et une méthode qui modifie les valeurs de l'objet `Personne`.

5

Applications, Fichiers et Gestionnaire de projets

Lors de cette journée, nous examinerons les différents éléments qui composent une application Delphi, ou plutôt, un projet Delphi. En bout de chaîne, l'application Delphi sera un exécutable ou un groupe d'exécutables et de DLL. Si vous avez utilisé des contrôles ActiveX, ces derniers seront intégrés eux aussi. Les contrôles ActiveX sont des contrôles OLE que vous pouvez écrire en Delphi, en C++ et dans quelques autres langages, et utiliser dans vos projets. Ces contrôles ne sont pas pour autant intégrés à votre exécutable. Vous en apprendrez plus au sujet des contrôles ActiveX dans le Chapitre 13, "Créer des Composants Visuels". Pour le moment, ce qui nous intéresse vraiment est de savoir quels sont précisément les fichiers, fiches, formes, unités, composants, ressources, etc. qui sont construits par vous ou par Delphi.

Composition d'une application Delphi

A première vue, un programme simple semble n'être composé que d'un fichier de projet et d'une unité. En réalité, d'autres fichiers sont créés pour vous en coulisse à mesure que vous travaillez sur votre programme. Pourquoi se soucier des fichiers créés par Delphi ? Imaginez un peu votre tête si vous effaciez par mégarde un fichier indispensable à l'application critique sur laquelle vous vous échinez depuis des semaines... Dans tous les cas de figure, il n'est jamais inutile de bien comprendre ce qui compose un projet Delphi et de savoir d'où viennent tous ces fichiers supplémentaires que vous n'avez pas créés, et ce qu'ils font.

Cette partie aborde plusieurs sujets importants, qui pour certains mériteraient qu'on leur consacre une journée entière. Mais nous ne ferons que tracer les grandes lignes d'un projet Delphi. Si vous avez besoin de plus de détails concernant un sujet particulier, vous pouvez vous reporter à l'aide en ligne, aux manuels Delphi, ou à d'autres leçons de ce livre. Cette section se propose de vous donner une vue d'ensemble, facilement assimilable.

Une fois que vous saurez ce qui constitue un projet Delphi, vous serez à même de gérer des projets. L'EDI de Delphi comprend un Gestionnaire de projet qui vous assiste dans cette tâche. Cependant, la gestion de projet ne se limite pas à l'utilisation d'un menu. Si vous désirez un projet organisé et bien géré, vous devez penser à définir une structure de répertoires apte à stocker votre code, en utilisant des noms appropriés pour les fichiers, les fiches, les composants et les variables. La meilleure méthode consiste à vous organiser dès le début et à vous tenir à cette organisation jusqu'à l'achèvement du projet.

Projets

Un projet Delphi est constitué de fiches, d'unités, de paramètres d'options, de ressources, etc. Toutes ces informations résident dans des fichiers. La plupart de ces fichiers sont créés par Delphi à mesure que vous construisez votre application. Les ressources telles que les bitmap, les icônes, etc. se trouvent dans des fichiers provenant de sources tierces ou sont créées avec les nombreux outils et éditeurs de ressources dont vous disposez. De plus, des fichiers sont également créés par le compilateur. Jetons un bref coup d'œil sur ces fichiers.

Nouveau

Un projet Delphi est constitué de fiches, d'unités, de paramètres d'option, de ressources, etc.

Les fichiers suivants sont créés par Delphi à mesure que vous concevez votre application :

- Le fichier projet (.dpr). Ce fichier stocke des informations concernant les fiches et les unités. Le code d'initialisation se trouve aussi là.
- Le fichier d'unités (.pas). Ce fichier stocke du code. Certaines unités sont associées à des formes, d'autres se contentent de stocker des fonctions et des procédures. La plupart des fonctions et procédures de Delphi se trouvent dans des unités.
- Fichier de fiches (.dfm). Ce fichier binaire est créé par Delphi pour stocker des informations concernant vos fiches. A chaque fiche correspond un fichier Unit (.pas). Ainsi, à mafiche.pas est associé un fichier mafiche.dfm.
- Fichier d'options de projet (.dfo). Les paramètres d'option du projet sont stockés dans ce fichier.
- Fichiers d'informations de paquet (.drf). Ce sont des fichiers binaires utilisés par Delphi pour la gestion des paquets.
- Fichier de ressources (.res). Ce fichier binaire contient une icône utilisée par le projet. Ce fichier ne doit pas être créé ou modifié par l'utilisateur. Delphi met à jour ou recrée constamment ce fichier.
- Fichiers de sauvegarde (.~dp, .~df, .~pa). Ce sont des fichiers de sauvegarde pour les fichiers de projet, de fiches et d'unités, respectivement.

Les fichiers suivants sont créés par le compilateur.

- Fichier exécutable (.exe). C'est le fichier exécutable de votre application. Il s'agit d'un fichier exécutable indépendant qui n'a besoin de rien d'autre que lui-même, sauf si vous utilisez des bibliothèques contenues dans des DLL, VBX ou autres.
- Fichier d'objet unité (.dcu). Ce fichier est la version compilée des fichiers d'unités (.pas) et sera lié dans le fichier d'exécutable final.
- Bibliothèque de liaison dynamique (ou DLL) (.dll). Ce fichier est créé si vous concevez vos propres DLL.

Enfin, voici d'autres fichiers Windows qui peuvent être utilisés avec Delphi.

- Fichiers d'aide (.hlp). Ce sont des fichiers d'aide Windows standard qui peuvent être utilisés avec votre application.
- Fichiers graphiques ou d'images (.wmf, .bmp, .ico). Ces fichiers sont fréquemment utilisés dans les applications Windows pour leur donner un aspect agréable et convivial.

Nouveau

Le fichier de projet (.dpr) lui-même contient en fait du code Pascal Objet et constitue la partie principale de votre application qui lance les choses lorsque vous exécutez votre application. Ce qui est amusant, c'est que vous pouvez construire une application Delphi sans jamais être obligé de voir ce fichier. Il est créé et modifié automatiquement par Delphi à mesure que votre application se construit. Le nom que vous donnez à votre fichier de projet sera aussi celui de votre fichier exécutable. Le code ci-après montre à quoi ressemblerait un fichier de projet si vous commenciez un projet sans changer les noms des fichiers ou des fiches.

```
program Project1
uses
Forms,
Unit1 in 'UNIT1.PAS' {Form1};

{$R *.RES}

begin
Application.CreateForm(TForm, Form1);
Application.Run(Form1);
end.
```

Info *Le mot program à la première ligne indique à Delphi qu'il s'agit du code du programme principal. Program sera remplacé par library (bibliothèque) si vous construisez une DLL.*

Attention *Ce fichier est géré automatiquement par Delphi. Il n'est pas recommandé de modifier un fichier de projet, à moins de vouloir écrire une DLL ou de faire de la programmation avancée.*

Cependant, vous pouvez si vous le désirez voir le code source du projet en sélectionnant Voir | Source du projet. Vous en apprendrez plus à ce sujet au Jour 6.

Fiches

Le mieux, dans un programme Windows, c'est bien la fiche. Avant de commencer à écrire des programmes Windows, nous programmions surtout des applications DOS en C. C est un très bon et très puissant langage, mais il est loin d'être facile ou souple dès qu'il s'agit de s'atteler à la programmation Windows. Nous avons alors suivi des cours de C++, puis des cours de programmation Windows, et nous avons commencé à programmer en C/C++. Il nous fallait des heures pour créer une simple fiche comportant un bouton, et plus d'heures encore pour comprendre ce que nous venions de faire et comment fonctionnait le code. La quantité de code source nécessaire à la création d'une simple fenêtre en C est gigantesque.

En C++, les choses sont un peu plus faciles si vous utilisez un kit de développement, comme OWL (Object Windows Library) de Borland ou MFC (Microsoft Foundation Class) de Microsoft, mais vous avez tout de même besoin d'une solide connaissance de la programmation orientée objet (voir le Jour 4 à ce sujet) et du langage lui-même pour savoir ce que vous faites et ce que vous pouvez faire. Même les compilateurs C++ les plus récents ne proposent pas la programmation visuelle à laquelle excelle Delphi. Delphi donne la possibilité de créer de vrais programmes Windows qui détrônent certaines applications développées dans d'autres environnements visuels. Les programmes Delphi tourneront à des vitesses proches de celles des programmes C++. Delphi se charge de la plus grosse partie du travail et vous laisse vous consacrer à la partie de code la plus spécifique à votre application.

Nouveau

Comme vous le savez, un programme Windows est constitué d'une fenêtre ou d'un ensemble de fenêtres, appelées fiches dans Delphi. Lorsque vous démarrez Delphi, il crée automatiquement une fiche à votre usage. Les fiches accueillent vos contrôles, composants, etc.

L'application Delphi (comme du reste la plupart des applications Windows) est centrée autour de la fiche. Bien que d'autres programmes puissent utiliser le concept de fenêtres ou de fiches, pour qu'une application soit entièrement conforme à l'esprit de Microsoft Windows, elle doit respecter les spécifications de Microsoft dans sa disposition et la structure de son aspect. Les informations concernant les fiches Delphi sont stockées dans deux fichiers : les fichiers .dfm et .pas. Le fichier .dfm contient en fait des informations sur l'apparence, la taille, l'emplacement de votre fiche. Vous n'avez pas besoin de vous préoccuper de ce fichier, Delphi s'en charge, mais il n'est pas inutile que vous sachiez quelle est sa fonction.

Le code de la fiche et le code des contrôles qu'elle contient sont stockés dans le fichier .pas, aussi appelé *unité*. C'est le fichier sur lequel vous passez le plus de temps lorsque vous écrivez une application Delphi. Chaque fois que vous ajoutez un gestionnaire d'événements pour une fiche, ou que vous double-cliquez sur un contrôle pour y ajouter du code, le fichier .pas est mis à jour et Delphi place le curseur à l'emplacement adéquat, pour que vous complétiez ou modifiiez votre code. Lorsque vous ajoutez d'autres fiches, elles possèdent elles aussi leurs propres fichiers .dfm et .pas.

Une autre chose à savoir concernant les fiches est qu'elles ont des propriétés. Ces propriétés peuvent être définies pour contrôler l'aspect et le comportement de la fiche. Grâce à ces propriétés, vous pouvez modifier la couleur, la taille, l'emplacement de la fiche, indiquer si elle est centrée, placée à un endroit précis, visible, invisible, etc. Une forme comporte aussi un certain nombre de *gestionnaires d'événements* (segments de code s'exécutant lorsque des événements spécifiques liés à la fiche surviennent). Vous pouvez inclure des gestionnaires d'événements pour des événements tels qu'un clic de souris ou un redimensionnement de fiche.

Note

Nous avons pratiquement tout dit des fichiers qui composent un projet Delphi. Pour la plupart d'entre eux, ces fichiers seront synchronisés par Delphi lorsque vous procéderez à vos mises à jour. Il est recommandé de toujours utiliser Delphi pour changer le nom des fichiers ou pour les mettre à jour, afin d'éviter que les fichiers ne soient désynchronisés. Si vous passez outre, vous risquez de provoquer des erreurs lors du chargement ou de la compilation de vos programmes. Autrement dit, si vous cliquez sur un composant avant de le supprimer, laissez Delphi supprimer lui-même le code associé. Ne le supprimez pas vous-même dans l'éditeur, Delphi se charge très bien de faire le ménage.

Unités

On compte trois types d'unités : les unités associées à des fiches (elles sont les plus courantes), les fichiers d'unités qui stockent les fonctions et procédures, et les fichiers d'unités permettant de construire des composants.

Nouveau

Les fichiers d'unités sont des fichiers de code source comportant l'extension .pas. En travaillant avec Delphi, vous utiliserez sans cesse des unités.

Examinons ensemble une unité simple associée à une fiche (voir Listing 5.1). Le nom de l'unité figure à la première ligne qui suit le mot `unit`. Au dessous de l'en-tête `unit` ce trouve la partie `interface` qui contient les clauses `uses`, `type` et `var`. Enfin, la partie implémentation contient les fonctions et procédures de vos contrôles (les gestionnaires d'événements), de même que vos propres fonctions, procédures, et de manière générale tout le code qui sera utilisé dans l'unité. La partie `Implementation` peut également contenir une clause `uses`.

Listing 5.1 : Unité1

```
unit Unite1;

interface

uses
 Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin

end;
end.
```

Le code ci-avant, accompagné du code du fichier de projet, suffit pour créer un exécutable Delphi qui ouvre une fenêtre. Le programme ne fera pas grand-chose de plus, mais c'est un programme Windows fonctionnel dans sa forme la plus simple. Ce code est le code créé par défaut

par Delphi lorsque vous commencez un nouveau projet. Il ne se compilera pas si vous le tapez vous-même dans l'éditeur de code sans créer de nouveau projet.

Regardez les noms qui figurent dans la clause `uses`. Ce sont les noms d'autres unités. Si vous décidez d'écrire une série de fonctions et de procédures utiles, vous pouvez créer votre propre unité, y placer tous vos utilitaires, et compiler cette unité pour en faire usage ultérieurement. Chaque fois que vous souhaitez utiliser votre unité faite maison, il vous suffit d'ajouter son nom dans la clause `uses`. Examinons ensemble les différentes parties qui composent l'unité du Listing 5.1.

- L'en-tête de l'unité. Un en-tête d'unité identifie le code comme une unité, et il est suivi du nom, et de celui du fichier de l'unité, dont l'extension sera .pas.
- `interface`. Cette clause marque le début de la partie interface de l'unité, dans laquelle sont déclarés variables, types, procédures, etc. La partie interface détermine ce qui dans cette unité est disponible pour les autres unités et parties du programme. La partie interface s'achève pour laisser la place à la partie implémentation.
- `uses`. La clause `uses` indique au compilateur quelles sont les bibliothèques de fonctions et de procédures qui doivent être compilées dans l'exécutable final. Delphi en inclut certaines automatiquement. Si vous écrivez votre propre unité, vous devez vous souvenir d'en inclure le nom dans la clause `uses` lorsque vous avez besoin des fonctions qu'elle contient.
- `type`. La partie déclaration de type permet de créer les types définis par l'utilisateur. Ces types peuvent ensuite être utilisés pour définir des variables.

  Les spécificateurs de visibilité suivent la clause type dans la partie interface (voir Listing 5. 1). Les spécificateurs suivants sont utilisés pour contrôler la visibilité d'un objet pour d'autres programmes ou objets.

- `Private`. Les déclarations dans cette section sont traitées comme publiques dans le module, mais resteront inconnues et inaccessibles en dehors de l'unité.
- `Public`. Les déclarations placées dans cette partie sont visibles et accessibles en dehors de l'unité.

  Les deux spécificateurs suivants sont utilisés pour la création de composants (ils ne sont pas nécessaires dans le Listing 5.1, et nous les traiterons en détail au Jour 13). Nous les mentionnons ici par souci d'exhaustivité.

- `Published`. Utilisé pour créer des composants. Les propriétés publiées sont affichées dans l'Inspecteur d'objets pour vous permettre de les modifier au cours de la conception.
- `Protected`. Un composant. Les champs, méthodes et propriétés déclarés comme protégées sont accessibles aux descendants (dont nous parlerons au Jour 4) du type déclaré (les types ont été décrits au Jour 3).

  Les quatre spécificateurs (`private`, `public`, `protected` et `published`) font partie de la définition de classe .

- `var`. Utilisé pour déclarer les variables et les variables des objets. Dans une unité de fiche, `var` est utilisé dans la partie `interface` (Delphi place cette déclaration pour vous) pour déclarer la fiche comme instance de l'objet `TForm`. `Var` est également utilisé pour déclarer les variables dans la partie d'implémentation ainsi que dans les procédures et les fonctions.
- `implementation`. C'est là que toutes les fonctions et procédures déclarées dans la partie interface seront placées. Toute déclaration faite dans cette partie est privée pour l'unité (elle n'est donc pas disponible pour les autres unités). Vous pouvez cependant ajouter une clause `uses` dans la section `implémentation` afin d'avoir accès à d'autres unités.
- {$R *.DFM}. Dans une unité de fiche, Delphi insère l'entrée $R *.DFM à votre place. Cette entrée est très importante car elle lie la forme à son fichier .dfm dont nous avons parlé tout à l'heure. Pour vous éviter des problèmes, *ne retirez pas* cette entrée de votre programme,

Le bloc de code suivant s'exécute lorsque votre fiche est créée. C'est là que vous devez placer le code de démarrage qui doit être exécuté lorsque la fiche commence à se charger. Pour créer cette procédure, utilisez l'Inspecteur d'objets pour visualiser le menu Evénements de la fiche, puis double-cliquez sur l'événement `OnCreate`. L'Inspecteur d'objets est décrit en détail au Jour 8.

```
procedure TForm1.FormCreate(Sender: TObject);
begin

end;
```

N'oubliez bien sûr pas d'ajouter le `end.` et remarquez le point qui le suit. Cela signifie qu'il s'agit de la fin de l'unité.

```
end.
```

Enfin, vous pouvez éventuellement placer deux autres parties dans votre unité. Reportez-vous à l'aide en ligne pour en savoir plus à ce sujet.

- `initialization` La partie d'initialisation est utilisée pour inclure le code permettant d'initaliser et de préparer l'exécution de votre unité.
- `finalization` La partie de finalisation vous permet de faire un peu de ménage avant que votre unité ne s'achève complètement.

Ici s'achève notre examen d'une unité associée à une fiche. Il est important que les en-têtes, les clauses et le reste se trouvent en ordre correct. Delphi se charge pour une bonne partie de tout cela puisqu'il crée l'unité et le code à mesure que vous apportez des modifications à la fiche. Il vous suffit d'ajouter votre code à la section adéquate. Nous ne parlerons pas ici des deux autres types d'unités, les unités associées à un composant et les unités qui stockent les fonctions et les procédures. L'unité qui stocke les fonctions et les procédures est très similaire à une unité de fiche qui ne comporterait pas de déclarations concernant les fiches. Pour plus de détails à ce sujet, consultez les manuels ou l'aide en ligne.

Bibliothèque des composants visuels (VCL)

La bibliothèque des composants visuels (VCL) est constituée d'une riche sélection d'objets écrits en Pascal Objet Delphi pour servir de contrôles (ou composants) dans les fiches Delphi. Ces composants prêts à l'emploi utilisent l'API de Windows et vous permettent d'interagir avec Windows sans connaître grand-chose du fonctionnement sous-jacent des API. Il existe deux types fondamentaux de composants : visibles et invisibles. Les composants visibles sont placés et programmés pour l'utilisateur. Les composants invisibles donnent un contrôle ou une interface de programmation spéciale à l'usage du programmeur. Un timer est un exemple de composant invisible tandis que la boîte d'édition est un exemple de composant visible.

Les différentes catégories de composants comprennent les boutons, les boîtes de listes, les libellés, les boîtes d'édition, les composants sensibles aux données, les timers, les boîtes à images, les boîtes de sélection, et bien d'autres encore. Vous apprendrez à connaître et à apprécier ces composants car ils forment les blocs de construction de votre application Delphi. Chacun de ces composants est associé à un code parfois conséquent qui assure son fonctionnement, et vous n'avez pas à écrire ce code. Il vous suffit de placer les composants sur la fiche pour qu'ils soient prêts à l'emploi. Les composants ont des propriétés, des événements et des méthodes qui permettent au programmeur de les utiliser et de contrôler leur comportement. Nous traiterons de la VCL plus en détail au Jour 8.

Composants ActiveX facultatifs

Info *Un contrôle ActiveX est un contrôle ou un composant Windows 32 bits, basé sur OLE et prêt à l'emploi, qui peut être utilisé par Delphi 3 et par tout autre langage prenant en charge ActiveX. La plupart du temps, les contrôles ActiveX sont développés par des programmeurs ou éditeurs indépendants, et créés en C++, mais Delphi, comme vous le verrez au Jour 13, permet de créer des contrôles ActiveX.*

Delphi 3 prend en charge les contrôles ActiveX. C'est une bonne et une mauvaise chose. L'avantage est qu'il existe sur le marché des myriades de contrôles ActiveX prêts à l'emploi. Vous pouvez trouver un ActiveX pour n'importe quel usage, des onglets d'agenda aux contrôles de communications dotés de toutes les fonctions imaginables. L'inconvénient des ActiveX est qu'ils ne sont pas compilés dans votre exécutable, et que vous devez en conséquence les fournir avec votre application. Il est important de toujours garder à l'esprit que, lorsque vous avez vraiment besoin d'un ActiveX, si vous trouvez celui qui fonctionne avec Delphi et répond à vos besoins, n'hésitez pas. La démarche conseillée passe cependant par l'utilisation de composants Delphi. Petit à petit, le nombre de composants Delphi proposés augmentera et vous n'aurez plus besoin d'ActiveX.

Les contrôles ActiveX sont les nouveaux venus dans le petit monde de la réutilisabilité des composants. Ils sont reconnus par la grande majorité des environnements graphiques de développement, et peuvent être utilisés sur l'Internet et dans des applications Web. Les contrôles ActiveX remplacent les contrôles OCX de Delphi 2. Vous en apprendrez plus à ce sujet au Jou 13.

Codes de procédures, de fonctions et de gestionnaires d'événements créés par l'utilisateur

Les procédures, les fonctions et les gestionnaires d'événements créés par l'utilisateur forment une partie importante de votre application. A mesure que vous construisez votre application, vous créez des fiches utilisant les composants, mais vous avez besoin à un moment ou à un autre de tout lier ensemble au moyen de votre propre code. Tous ces composants sont sans grande utilité si vous ne disposez pas d'un moyen pour y accéder afin d'obtenir des informations, ou pour leur indiquer ce qu'ils doivent faire. Le Pascal Objet qui sous-tend Delphi vous propose une large gamme d'outils pour construire vos programmes. De plus, vous ressentirez vite le besoin d'écrire vos propres routines en Pascal Objet.

Ressources graphiques

Il ne faut pas oublier ce qui fait tout le charme d'un programme : les graphismes. Vous ne pouvez construire une application un tant soit peu agréable sans y faire figurer quelques graphismes. Delphi est fourni avec quelques icônes et bitmap, mais vous ne tarderez pas à commencer votre propre collection. Lorsque vous choisissez une icône pour votre programme, Delphi la stocke dans le fichier .res. Les icônes ne sont pas cependant les seules ressources utilisées par un programme Delphi. Les curseurs, les bitmap et les métafichiers Windows (.wmf) sont également des ressources.

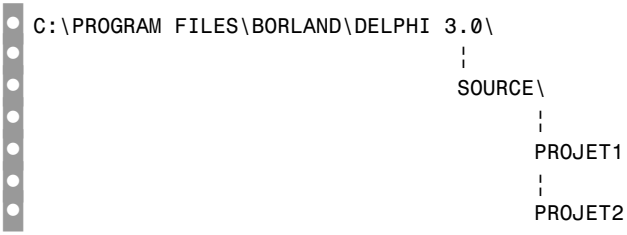
Tous ces objets deviennent partie intégrante de votre exécutable, mais ce sont à l'origine des ressources extérieures à votre code source. Vous choisissez généralement les ressources que vous désirez en définissant des propriétés dans un composant ou en spécifiant un nom de fichier. Ainsi, le composant charge le bitmap, l'icône ou tout autre graphique. Le graphique est ensuite chargé dans le projet, où il sera prêt à l'emploi lorsque vous sélectionnerez Exécuter ou Compiler.

Organiser son projet Delphi

Maintenant que vous savez de quoi sont constitués les projets Delphi, il est temps de penser à la façon de les organiser et de les gérer. Vous pouvez fort bien travailler sur de petits programmes ou projets simples sans trop vous soucier de gestion de projet, mais si vous n'y prenez pas garde, vous risquez de supprimer ou d'écraser des fichiers indispensables. Il est aussi très facile de ne plus très bien savoir si un fichier appartient à un projet ou à un autre, pour peu que vous ne soyez pas organisé. C'est pourquoi, dans le reste de cette leçon, nous allons voir ensemble comment prendre de bonnes habitudes d'organisation et s'y tenir. Cela ne demande pas beaucoup d'efforts, mais vous gagnerez du temps et vous épargnerez du travail si vous savez parfaitement où se trouvent tous les fichiers de votre projet quand vous voulez travailler sur certains d'entre eux, en supprimer, en sauvegarder, et plus généralement, les manipuler.

Créer des répertoires distincts

Commençons par le plus simple. Il est bien de créer un répertoire source sous le répertoire du compilateur. Sur notre ordinateur, ce répertoire est C:\program files\Borland\Delphi 3.0\source. En aval du répertoire source, nous créons en général un répertoire pour chacun de nos projets. Cela permet de trouver ou de sauvegarder un projet plus facilement, et cela garantit également que les fichiers ne seront pas modifiés accidentellement. Supposez qu'un de vos projets utilise une unité appelée mainmenu.pas et qu'un autre projet soit copié dans le même répertoire. Ce projet possède lui aussi un mainmenu.pas. Oui, vous avez deviné, le mainmenu.pas initial est perdu. Observez donc cette structure de répertoire :



Si vous mettez en place une structure de ce type, les projets ont peu de chance d'entrer en collision. Lorsque vous sélectionnez Fichier Ouvrir ou Fichier Enregistrer sous dans le menu Fichier de Delphi, le répertoire que vous avez sélectionné s'affiche. Un autre avantage de cette méthode est que vous savez toujours où vont vos fichiers et vos exécutables mis à jour (à supposer que vous ne vous soyez pas placé entre temps dans le mauvais répertoire ou que vous n'ayez pas demandé à Delphi d'enregistrer les fichiers ailleurs). Cette structure fait de la sauvegarde un jeu d'enfants. Vous pouvez utiliser l'Explorateur Windows, le Gestionnaire de fichier ou un utilitaire de sauvegarde pour effectuer une sauvegarde de C:\program files\borland\Delphi 3.0\source et de tous ses répertoires.

Conventions de nommage

Une autre bonne habitude à prendre pour mieux vous organiser (et pour savoir ce que sont vos fichiers source Delphi d'un simple coup d'œil) consiste à utiliser une convention de nommage pour vos noms de fichiers. Autrement dit, n'utilisez pas les noms fournis par défaut, tels que project1.dpr et unit1.pas. La meilleure manière de ne pas prendre de mauvaises habitudes est sans doute de faire immédiatement Projet Enregistrer sous lorsque vous démarrez un nouveau projet. Donnez aux fichiers unit1.pas et project1.pas de nouveaux noms lorsqu'il vous en est fait la demande. Bien entendu, chaque fois que vous ajoutez une fiche, vous devez lui donner un nouveau nom lorsque vous l'enregistrez. Toutefois, si vous nommez toutes les fiches lorsque vous commencez, vous pouvez de temps en temps enregistrer le projet sans devoir vous occuper de noms de fichiers et vous bénéficierez ainsi des avantages que nous venons d'évoquer. Si vous ne faites pas cela, imaginez un peu le casse-tête quand vous reviendrez 6 mois plus tard sur votre source et essaierez de deviner ce que peuvent bien faire unit1.pas et unit2.pas dans le projet project1.dpr...

En fait, une bonne convention de nommage devrait s'appliquer non seulement à vos fichiers, mais aussi aux fiches, aux composants et aux variables que vous utilisez dans votre application. On est facilement tenté d'utiliser les noms par défaut donnés aux fiches et aux composants au fur et à mesure de la conception. Cependant, une fois que vous avez donné aux fichiers des noms uniques, vous pouvez quand même vous retrouver avec un projet ingérable s'il est de taille importante. Imaginez votre projet, dans lequel figurent FORM1, FORM2, FORM3, chacune avec son lot de BUTTON1, BUTTON2, BUTTON3... Vous verrez souvent des exemples de ce genre de code, y compris dans ce livre. Quand il s'agit d'une seule fiche ne comportant que quelques composants, ce n'est pas trop grave. Mais si votre projet prend de l'ampleur, vous finirez par perdre du temps à regarder à l'intérieur du code pour voir ce que fait tel ou tel composant ou une fiche. En prenant l'habitude de donner des noms descriptifs (et uniques) à vos fichiers, fiches, composants, etc., vous gagnerez du temps et vous épargnerez des frustrations.

Pour mieux me faire comprendre, nous allons comparer deux listings. Le premier est un listing (Listing 5.2) qui crée une fiche à l'écran chargée de prendre le nom d'un utilisateur. Toutes les données par défaut pour les noms de fichiers, les titres et les propriétés de nom ont été modifiées pour être plus descriptives. Le second, le Listing 5.3, reprend le même code mais n'utilise que les noms par défaut affectés par Delphi lors de la création de l'unité.

Note *Les noms de fichier doivent suivre les règles de longueur et de caractères autorisés du système d'exploitation auquel l'application est destinée (Windows 95 ou Windows NT par exemple). Les titres ont une taille maximale de 255 caractères et doivent être des chaînes Pascal valides. Les noms ont une taille maximale de 63 caractères et doivent suivre les règles qui s'appliquent aux identificateurs Pascal (pour plus de détails sur les identificateurs, reportez-vous à l'aide en ligne).*

Listing 5.2 : Lirenom

```
unit Lirenom;

interface

uses
 Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs

type
  TEntreeNomUtilisateur = class(TForm)
    Bouton_EntreeNom: TButton;
    Edit_EntreeNom: TEdit;
    procedure FormCreate(Sender: TObject);
    procedure Bouton_EntreeNomClick(Sender: TObject);
    procedure Edit_EntreeNomChange(Sender: TObject);
  private
    { Déclarations publiques }
```

```
  public
    { Déclarations privées }
  end;

var
  EntreeNomUtilisateur: TEntreeNomUtilisateur;

implementation

{$R *.DFM}

procedure TEntreeNomUtilisateur.FormCreate(Sender: TObject);
begin

end;

procedure TEntreeNomUtilisateur.Bouton_EntreeNomClick(Sender: TObject);
begin

end;

procedure TEntreeNomUtilisateur.Edit_EntreeNomChange(Sender: TObject);
begin

end;

end.
```

Voyons maintenant un code identique, à ceci près que tous les noms par défaut ont été conservés tels quels (Listing 5.3).

Listing 5.3 : Unit1 (avec les noms par défaut)

```
unit Unit1;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs;

type
```

```
  TForm1 = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Edit1Change(Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin

end;

procedure TForm1.Button1Click(Sender: TObject);
begin

end;

procedure TForm1.Edit1Change(Sender: TObject);
begin

end;

end.
```

Le Listing 5.2 est bien plus facile à suivre car les noms décrivent avec précision la nature et la fonction de chaque objet. Prenons `Bouton_EntreeNom`, il est clair qu'il s'agit d'un bouton censé prendre un nom. Alors que `Button1`... Ces conventions de nommage fixées, vous êtes paré pour passer au Gestionnaire de projet afin d'apprendre à conduire un projet Delphi.

Exemple de projet

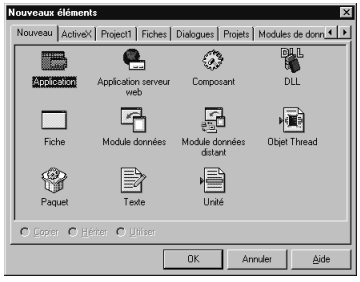
Nous allons construire un petit projet dans son intégralité. A mesure que nous progresserons, vous apprendrez à mettre en pratique les notions que nous avons présentées au début de cette journée. Vous découvrirez également le Gestionnaire de projet et d'autres fonctionnalités de Delphi qu'il vous est nécessaire de connaître pour travailler sur des projets. Vous allez créer une application Delphi simple qui comporte deux fiches interagissant l'une avec l'autre.

Pour bien démarrer, mettons en place une structure de répertoire bien organisée.

Commencez par créer les fichiers de projet :

1. Créez un répertoire source en aval du répertoire de Delphi (si ce répertoire source n'existe pas déjà), C:\program files\borland\Delphi 3.0\source par exemple.
2. Créez un répertoire dans lequel vous allez stocker votre projet. Appelons-le CauseFiche (C:\program files\borland\Delphi 3.0\source\causefiche). Si vous vous demandez la raison de ce nom, elle vient du fait que dans notre application, une fiche parlera à l'autre, ou plus précisément, elle mettra à jour les informations d'une autre fiche.
3. Si Delphi ne tourne pas déjà, lancez-le. Créez un nouveau projet en faisant Fichier/Nouveau, sélectionnez l'icône Application dans la boîte de dialogue Nouveaux éléments, puis cliquez sur OK (voir Figure 5.1).

Figure 5.1
Boîte de dialogue Nouveaux éléments avec l'icône Application en surbrillance.



4. Cliquez sur l'icône Fiche ou sélectionnez Fichier | Nouvelle fiche afin d'ajouter une seconde fiche au projet.
5. Maintenant, vous devez modifier certaines des propriétés de la fiche et des ses composants, toujours dans un souci d'organisation. Utilisez l'Inspecteur d'objets pour modifier la propriété `caption` (titre) pour qu'elle devienne Fiche d'entrée, et la propriété `name` en Fiche d'entrée.
6. A l'aide de l'Inspecteur d'objets, changez le nom du titre de la nouvelle fiche en Fiche de sortie et le nom en Fiche de sortie également (le titre correspond à la propriété `caption`, et le nom à la propriété `name`). Cette fiche est destinée à recevoir la sortie provenant de la

fiche d'entrée. Les données seront transférées vers cette fiche après entrée des données et appui sur le bouton d'envoi.

7. Enregistrez la fiche, en changeant unit1.pas pour un autre nom, comme indiqué Figure 5.3 :

   ```
   C:\program files\borland\Delphi 3.0\source\causefiche\sortie.pas
   ```

   Cette fiche sera la fiche de saisie destinée à demander des informations à l'utilisateur. Enregistrez ensuite `Unit2` sous le nom `sortie.pas`.

Figure 5.2
L'option de menu Fichier | Enregistrer projet sous.

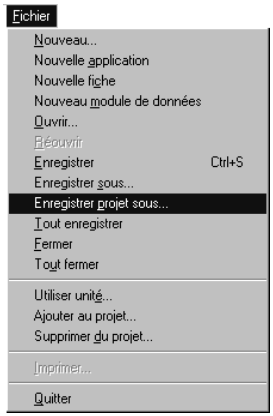
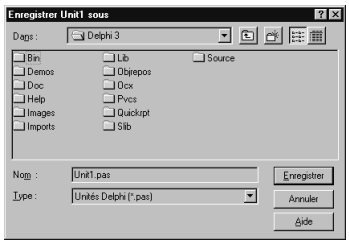


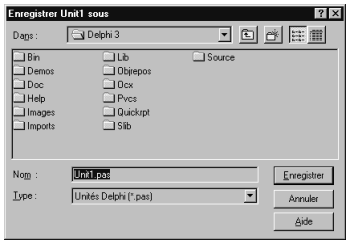
Figure 5.3
La boîte de dialogue Enregistrer sous.



8. Pour enregistrer le fichier projet, sélectionnez Fichier | Enregistrer projet sous (cf. Figure 5.2). Dans la boîte de dialogue qui apparaît alors (voir la Figure 5.4), sélectionnez le répertoire et le nom de fichier afin d'enregistrer le fichier de projet sous :

   ```
   C:\program files\borland\Delphi 3.0\source\causefiche\causefi-
   che.dpr
   ```

Figure 5.4
La boîte de dialogue Enregistrer projet sous.



A présent, construisez la fiche en procédant comme suit :

1. Ajoutez une boîte d'édition à chacune des fiches ainsi qu'un bouton sur la fiche d'entrée (les onglets et composants de la VCL sont traités en détail au Jour 8). Ensuite, à l'aide de l'Inspecteur d'objets, effacez le contenu de la propriété `text` dans chacune des deux boîtes d'édition.
2. Donnez à la boîte d'édition de la fiche d'entrée le nom `TexteEntree`, et à celle de la fiche de sortie le nom `TexteSortie`. Définissez le titre du bouton de la fiche d'entrée comme "Envoi" et définissez sa propriété de nom comme `EnvoyerTexte`.
3. Pour vous assurer que les deux fiches seront visibles, vous devez utiliser l'Inspecteur d'objets pour modifier la propriété `visible` de chacune des fiches ajoutées (à l'exception de la première fiche que vous créez lorsque vous démarrez un nouveau projet) en `True` (Vraie).
4. Ajoutons maintenant une ligne de code au bouton, afin de rendre notre programme fonctionnel. Double-cliquez sur le bouton Envoi et ajoutez cette ligne de code :

```
FicheSortie.TexteSortie.Text:=FicheEntree.TexteEntree.Text;
```

Si tout s'est bien passé, vous devriez être en mesure de lancer le programme et de le tester en procédant selon les étapes suivantes :

1. Cliquez sur l'icône Exécuter ou appuyez sur F9. Le message suivant apparaît dans une boîte de dialogue :

```
La fiche 'FicheEntree' référence la fiche 'FicheSortie' déclarée dans
l'unité 'Entree'
qui n'est pas dans la liste USES. voulez-vous l'ajouter?
```

4. Cliquez sur le bouton Oui.
5. Cliquez sur l'icône Tout Enregistrer, ou sélectionnez Fichier | Tout enregistrer. Le code nécessaire est ajouté automatiquement.
6. Relancez le programme (cliquez sur l'icône Exécuter ou appuyez sur F9). Vos fiches devraient apparaître à l'écran.
7. Testez le programme en saisissant du texte dans la zone d'édition de la fiche de saisie, puis en cliquant sur le bouton Envoi.

Regardons maintenant à quoi ressemble notre programme. Le Listing 5.4 contient le code source qui est créé et stocké dans votre fichier de projet. Utilisez Voir/Source du projet pour visualiser le contenu du fichier de projet.

Listing 5.4 : CauseFiche

```
program CauseFiche;

uses
  Forms,
  Entree in 'ENTREE.PAS' {FicheEntree},
  Sortie in 'SORTIE.PAS' {FicheSortie};

{$R *.RES}

begin
  Application.CreateForm(TFicheEntree, FicheEntree);
  Application.CreateForm(TFicheSortie, FicheSortie);
  Application.Run;
end.
```

La clause `uses` inclut l'unité `Forms` (qui est le code Delphi permettant de créer des fenêtres), et les unités `Entree` et `Sortie` qui contiennent le code des fiches que vous avez créées.

Après le premier `begin`, vous vous pouvez remarquer les déclarations de méthode `Application.Initialize` et `Application.CreateForm`. Ces méthodes exécutent le code nécessaire au lancement et à la création de vos fiches. Pour simplifier, les méthodes sont des sections de code contenues dans un objet (notre fiche en l'occurrence), qui sont exécutées lorsque la méthode est appelée, comme dans notre exemple. Nous reviendrons plus en détail sur les méthodes et leur utilisation au Jour 8.

La dernière déclaration est Application.Run. Ceci lance notre application et exécute le code associé à nos fiches (n'oubliez pas qu'une partie du code est ajoutée par Delphi, et l'autre partie par vous). La source du projet est mise à jour par Delphi à votre place lorsque vous ajoutez ou supprimez des fiches.

A présent, double-cliquez sur la fiche `Entree` afin de voir le code ci-après (voir Listing 5.5). Vous pouvez remarquer à quel point le code est agréable à lire grâce aux noms descriptifs que vous avez choisis.

Listing 5.5 : Unité Entrée

```
unit Entree;

interface
```

```
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  OutPut, StdCtrls;

type
  TFicheEntree = class(TForm)
    TexteEntree: TEdit;
    EnvoiTexte: TButton;
    procedure EnvoiTextClick (Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  FicheEntree: TFicheEntree;

implementation

{$R *.DFM}

procedure TFicheEntree.EnvoiTextClick(Sender: TObject);
begin
     FicheSortie.TexteSortie.Text:=FicheEntree.TexteEntree.Text;
end;
```

Regardez le code de la fiche `Sortie` (Listing 5.6). La lecture du code des fiches `Entree` et `Sortie` devrait être facile grâce à l'utilisation de noms descriptifs pour les fichiers, les titres et les noms. Imaginez un projet complexe comportant de nombreuses fiches et contrôles. Avec les bonnes habitudes que vous avez prises, ce projet complexe vous posera beaucoup moins de problèmes. Grâce aux noms qui donnent une bonne idée de la nature et de la fonction des divers fiches et composants, il vous sera facile de naviguer même dans le plus imposant des projets.

Le code de cette application réside dans l'unité `Entree`. La clause `uses` de la section `implementation` de l'unité `Entree` comporte une référence à l'unité `Sortie`. Ceci permet d'accéder au code contenu dans cette unité. L'application fonctionne en exécutant la ligne de code suivante lorsqu'on appuie sur le bouton Envoi :

```
FichierSortie.TexteSortie.Text := FicheEntree.TexteEntree.Text;
```

Listing 5.6 : Unité Sortie

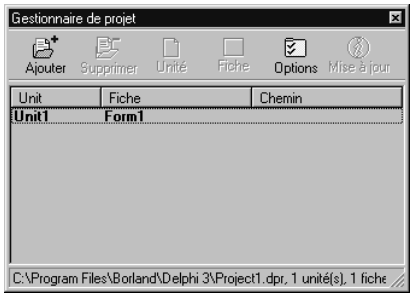
```
unit Sortie;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TFicheSortie = class(TForm)
    TexteSortie: TEdit;
  private
    { Déclarations privées}
  public
    { Déclarations publiques }
  end;

var
  FicheSortie: TFicheSortie;

implementation

{$R *.DFM}

end.
```

Ce code a été créé par Delphi lorsque vous avez créé la fiche `Sortie`. Comme cette fiche ne nous sert qu'à afficher les données envoyées par la fiche `Entree`, elle ne contient pas de code créé par l'utilisateur. Les sections de code de cette unité devraient vous être maintenant familières.

Gestionnaire de projet

Maintenant que vous avez un projet sur lequel travailler, voyons ensemble les fonctions offertes par le Gestionnaire de projet et les autres options de Delphi. Une fois un projet chargé, ouvrez le Gestionnaire de projet. Sélectionnez Voir/Gestionnaire de projet pour faire apparaître l'interface utilisateur du Gestionnaire de projet (Figure 5.5).

Figure 5.5
Le Gestionnaire de projet.



Dans l'interface du Gestionnaire de projet, vous pouvez remarquer que le nom de chaque unité figure dans la liste (`Entree` et `Sortie` dans notre exemple), ainsi que le nom de la fiche et le chemin d'accès (qui est vide pour l'instant puisque vous ne l'avez pas encore défini). Au-dessous de la barre de titre se trouve une barre d'icônes comportant quelques fonctions utiles.

Observez bien les options proposées par la barre d'icônes. Comme la plupart de ces options affectent l'unité sélectionnée, assurez-vous que vous sélectionnez la bonne unité avant de cliquer sur un des boutons de la barre d'icônes.

Ajouter unité. Vous permet d'ajouter un fichier d'unité. Ceci est bien pratique car vous pouvez vous retrouver avec une fiche conçue pour un autre programme et qui s'avère tout à fait adaptée au projet en cours. Il vous suffit alors d'ajouter la fiche au projet. Lorsque vous cliquez sur le bouton Ajouter fiche, la boîte de sélection de fichiers standard apparaît. Il ne vous reste plus qu'à parcourir les répertoires et à sélectionner le fichier de fiches à ajouter.

Supprimer unité. Supprime du projet l'unité sélectionnée (le fichier n'est pas supprimé, mais simplement retiré du fichier de projet).

Voir unité. Vous permet de voir le code de l'unité sélectionnée.

Voir fiche. Affiche la fiche associée à l'unité sélectionnée.

Options... Fait apparaître les Options du projet, une boîte de dialogue à onglets proposant diverses options (nous en reparlerons dans la section qui suit).

Mise à jour. Ce bouton est désactivé si vous n'avez pas modifié manuellement le fichier de projet. Le bouton permet de synchroniser le fichier de projet avec les fiches et unités listées dans le Gestionnaire de projet.

Attention

Il n'est pas recommandé de modifier le fichier source de projet à moins que vous ne soyez en train d'écrire une DLL ou que vous ne vous adonniez à de la programmation avancée. Delphi se charge en temps normal de la plupart des modifications à effectuer.

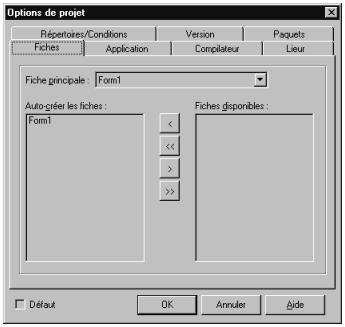
Comme vous pouvez le voir, le Gestionnaire de projet vous permet de naviguer rapidement dans votre projet, d'y ajouter ou d'y supprimer des fichiers, et d'aller dans la boîte de dialogue Options.

En plus du Gestionnaire de projet, vous utiliserez les menus Fichier, Voir, Projet, Groupe de travail (qui n'est disponible que si vous installez PVCS, dont nous parlerons brièvement à la fin de cette leçon) et Outils, ainsi que la barre d'icônes (qui vous permet d'accéder à des fonctions concernant le projet) lorsque, travaillant sur votre projet, vous aurez besoin de créer, de supprimer, de fermer et plus généralement de manipuler des fichiers de projet. Nous allons donc nous intéresser aux options et fonctions qui se rapportent directement à la gestion d'un projet.

Options de projet

La boîte de dialogue à onglets Options de projet vous permet de définir de nombreuses options concernant votre projet. Ces informations seront enregistrées dans le fichier .dfo dont nous avons parlé précédemment. Notre boîte de dialogue comprend sept onglets, comme le montre la Figure 5.6. Ces sections sont Fiches, Applications, Compilateur, Lieur, Répertoires/Conditions, Version et Paquets. Examinons ces onglets et les options qu'ils renferment. Vous pourrez par la même occasion vérifier les paramètres de votre projet CauseFiche.

Figure 5.6
La boîte de dialogue Options de projet (onglet Fiches).



Onglet Fiches

Le premier onglet est l'onglet Fiches (voir Figure 5.6). C'est l'onglet qui est actif lorsque vous ouvrez la boîte de dialogue Options de projet. Au sommet, vous trouvez la liste Fiche principale. Cette liste vous permet de sélectionner la fiche principale de votre menu. La fiche principale démarre en premier avec le code de son événement `OnCreate`. En temps normal, la première fiche que vous créez est la fiche principale. Pour ce projet, ce sera `FicheEntree`. Si vous cliquez sur la flèche vers le bas de la liste déroulante, vous verrez que vous pouvez choisir `FicheSortie` (qui est la seule autre fiche dans notre projet).

Dans l'onglet Fiches se trouvent également les boîtes de liste Auto-créer fiches et Fiches disponibles. Par défaut, les fichiers apparaissent dans la boîte de liste Auto-créer fiches. Les fiches disponibles font partie de votre projet mais devront être déplacées dans la section auto-créées

si vous souhaitez qu'elles soient créées au lancement. Dans le cas contraire, vous devez les activer au moment de l'exécution avant qu'elles ne soient utilisées. Lorsqu'une forme est ajoutée à la liste Auto-créées, le code adéquat est ajouté au fichier de projet sous la forme d'une déclaration telle que

```
Application.CreateForm(TFicheEntree, FicheEntree);
```

Note

Vous ne pouvez pas référencer une fiche tant qu'elle n'a pas été créée.

Pour déplacer des fiches d'une boîte de liste à une autre, vous pouvez utiliser les contrôles (les boutons sur lesquels figurent une flèche simple et une flèche double) qui se trouvent entre les deux boîtes. Vous pouvez également glisser-déplacer les fichiers entre les deux boîtes ou modifier leur ordre.

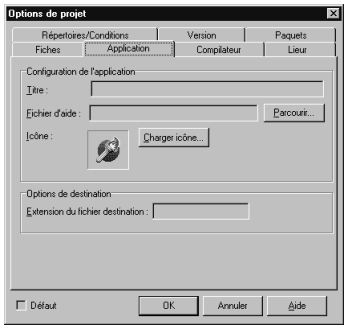
Si vous jetez un œil au projet sur lequel vous avez travaillé (`CauseFiche`), `FicheEntree` devrait être la fiche principale, `FicheEntree` et `FicheSortie` devraient toutes deux se trouver dans la boîte de listes Auto-créées et la boîte de liste Fiches disponibles devrait être vide. Une fois que `CauseFiche` aura été débogué et exécuté, amenez `FormeSortie` dans la boîte de liste fiches disponibles, puis exécutez l'application. Entrez **test** dans la boîte de texte de la fiche d'Entree et cliquez sur le bouton Envoi. Une erreur doit survenir. En effet, votre application n'a pas trouvé la fiche de `Sortie`. Replacez `FicheSortie` dans la boîte de listez Auto-créées et relancez le programme. Votre application doit maintenant fonctionner normalement.

Enfin, la case à cocher Défauts permet de faire des options courantes les options par défaut de tout nouveau projet. Vous ne cocherez pas cette case sauf si vous souhaitez que tous vos projets à venir utilisent les mêmes paramètres.

Onglet Application

Cette section vous permet de définir le titre, l'icône et le fichier d'aide de l'application, comme le montre la Figure 5.7. Toutes ces rubriques sont en principe vierges lorsque vous venez dans cet écran pour la première fois. Ces champs sont facultatifs, mais remplissez-les au fur et à mesure que vous lisez ce qui suit.

Figure 5.7
L'onglet Application.



Tout d'abord, le titre de l'application. C'est le texte que Windows affiche à côté de l'icône lorsque l'application est réduite. Sous Windows 95 et Windows NT 4.x, il apparaît dans la barre de tâches. Si vous laissez vide ce champ, l'application affichera le nom du projet. Ajoutez le nom Application Fiches bavardes. Ceci ajoutera la ligne suivante au code source du projet. Utilisez Voir/Source du projet pour voir cette ligne :

```
Application.Title := 'Form Talk Application';
```

Exécutez le projet et réduisez-le. Vous pouvez remarquer que Applications fiches bavardes est le nom de l'application qui apparaît à gauche de l'icône sur la barre des tâches de Windows 95. Si vous supprimez le texte dans l'écran du projet et exécutez celui-ci, vous verrez apparaître `CauseFiche` à la place. Assurez-vous que Application Fiches bavardes figure bien dans la boîte de texte avant de passer à la suite.

Juste au-dessous de la boîte de texte Titre de l'application se trouvent la boîte de texte Fichier d'aide et le bouton Parcourir. Vous pouvez sélectionner un fichier d'aide Windows standard pour l'associer à votre application. Essayez donc, en suivant ces étapes :

1. Cliquez sur Parcourir.
2. Placez-vous dans le répertoire \program files\borland\Delphi 3.0\help (selon la façon dont vous avez installé Delphi, les noms de répertoires peuvent être différents sur votre machine) et sélectionnez un des fichiers d'aide .HLP (delphi.hlp par exemple).
3. Visualisez de nouveau le code source. Vous pouvez voir que la ligne suivante a été ajoutée :

```
Application.HelpFile := 'c:\program files\borland\Delphi 3.
0\help\delphi .hlp;
```

Même s'il n'a pas grand-chose à voir avec votre programme, le fichier d'aide est associé au projet. Si vous configuriez correctement les numéros d'aide contextuelle dans les fiches et composants, il suffirait d'appuyer sur F1 pour faire apparaître l'aide Delphi associée au contexte défini pour la fiche ou le composant sélectionnés. Cette aide Delphi ne traite certes pas de votre application, mais c'est le principe de la configuration du fichier d'aide qui est important ici.

Ensuite se trouve une boîte d'image contenant l'icône par défaut ou celle déjà sélectionnée, qui apparaît sous la boîte d'édition Fichier d'aide et qui est suivie par un bouton Charger l'icône. Ce bouton Charger l'icône fait apparaître une boîte de sélection de fichier vous permettant de choisir l'icône de l'application. Cette icône est affichée lorsque vous réduisez la fiche principale de l'application. Cliquez sur le bouton Choisir icône, puis sélectionnez

```
C:\program files\borland\Delphi 3.0\images\icons\chip.ico
```

et cliquez sur OK. Vous ferez ainsi apparaître un microprocesseur dans la boîte d'image.

Exécutez de nouveau l'application. Réduisez-la, et vous devriez voir la nouvelle icône (représentant un microprocesseur) ainsi que le titre Application fiches bavardes. Faites Fichier/Enregistrer projet.

Note

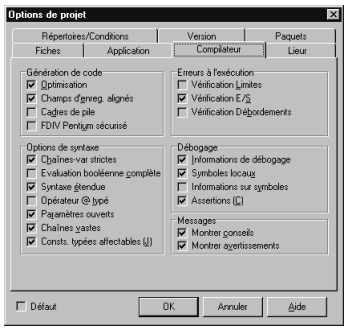
Vous n'êtes pas obligé de définir les options précédentes, mais elles sont recommandées car elles donnent un aspect plus "fini" à vos applications. La case à cocher permet d'indiquer que les options courantes deviendront les options par défaut pour tous les projet ultérieurs.

Enfin, au bas de l'onglet, dans la zone Options de Destination, vous pouvez préciser l'extension du fichier destination, par exemple `.OCX` pour un contrôle ActiveX.

Onglet Compilateur

Les sections de cet onglet (voir Figure 5.8) comprennent Génération de code, Erreurs d'exécution, Options de syntaxe et Options debug. Chacune de ces sections comprend un certain nombres d'options que vous pouvez activer ou désactiver. Pour la plupart des projets, les paramètres par défaut conviennent parfaitement. La case à cocher permet d'indiquer que les options courantes deviendront les options par défaut pour tous les projet ultérieurs.

Figure 5.8
L'onglet Compilateur.



Info

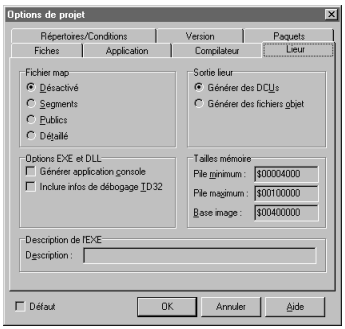
L'onglet Compilateur

Les options du compilateur ne rentrent pas dans le cadre de notre ouvrage. Cependant, à titre d'exemple, vous pouvez remarquer que sous les options de Génération de code figure une option appelée FDIV sécurisé Pentium. Cette option compile un code qui détecte la présence éventuelle d'un microprocesseur Pentium comportant le bogue de division en virgule flottante, et empêche que ce bogue ne se manifeste. Si vous savez que votre application ne sera pas exécutée sur des systèmes Pentium bogués, il vous suffit de désactiver cette option et votre code sera un peu plus rapide et compact. Nous vous recommandons cependant de laisser cette option activée, car il existe sûrement des gens qui n'ont jamais profité de l'offre de remplacement des microprocesseurs défectueux par Intel. Vous ne perdrez presque rien à laisser Delphi contourner un éventuel bogue. Cette option est mentionnée pour vous donner une idée des implications des options de compilateur, car elles affectent la taille de l'exécutable et la rapidité de votre application. Pour plus de détails, vous pouvez vous reporter aux manuels Delphi ou à l'aide en ligne de Delphi à laquelle vous accédez en appuyant sur F1 lorsque vous vous trouvez dans cet onglet.

Onglet Lieur

Les options du Lieur (voir Figure 5.9) comprennent Fichier map, Sortie lieur, options EXE et DLL, Tailles mémoire et Description de l'EXE. Comme dans le cas de l'onglet Compilateur, les options de cet onglet dépassent le cadre de ce livre. Pour plus de détails sur ces options, reportez-vous aux manuels Delphi et à l'aide en ligne. Là aussi, les options par défaut conviendront pour la plupart des projets.

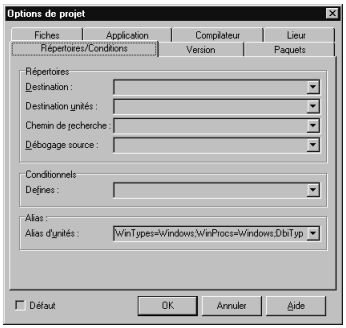
Figure 5.9
L'onglet Lieur.



Onglet Répertoires/Conditions

Dans la partie Répertoires (voir Figure 5.10), se trouvent trois groupes d'options : Répertoires, Conditionnels et Alias.

Figure 5.10
L'onglet Répertoires/ Conditions.



Le groupe Répertoires contient le répertoire de destination et les chemins de recherche. Le répertoire de destination vous permet d'indiquer l'emplacement où vous souhaitez que soient placés les exécutables (.exe). Destination unités vous permet d'indiquer où vous voulez placer les unités compilées (.dcu). Nous avons pour habitude de ne rien indiquer dans ces deux champs et de déplacer les fichiers compilés dans un autre répertoire une fois le travail fini.

Lorsque cette boîte de texte est laissée vide, les fichiers .dcu et .exe sont stockés dans le même répertoire que le code source.

Le chemin de recherche vous permet de spécifier là où seront recherchés les fichiers .dcu. Lorsque vous compilerez le programme, le compilateur ne recherchera que dans le chemin de recherche défini ici, dans le chemin de recherche de bibliothèque et dans le répertoire courant. Vous pouvez indiquer plusieurs entrées de chemins d'accès en les séparant par un point-virgule (;), chacun de ces chemins d'accès ne devant pas dépasser 127 caractères de long. Vous utilisez l'option Chemin de recherche si vous souhaitez que votre projet utilise des fichiers qui se trouve dans un autre répertoire que le répertoire courant ou le chemin de recherche de bibliothèque. Une erreur surviendra si les fichiers sources sont introuvables au moment de la compilation.

La zone Débogage source vous permet de spécifier l'emplacement de recherche des fichiers quand ils ont été déplacés depuis la dernière compilation. Il est inutile en règle générale de modifier cette option. Pour plus de renseignements, consultez l'aide en ligne.

La boîte de liste Conditions vous permet d'entrer ou de sélectionner dans une liste des directives et des symboles conditionnels de compilation (voir Figure 5.10). Ces symboles affectent le déroulement de la compilation, en fonction de certaines conditions. Pour plus de détails sur les directives et symboles de compilation, reportez-vous aux manuels Delphi ou à l'aide en ligne. La case à cocher permet d'indiquer que les options courantes deviendront les options par défaut pour tous les projet ultérieurs.

La boîte Alias vous permet d'entrer des alias pour d'autres unités. Vous pouvez remarquer l'apparition de l'entrée suivante :

```
WinTypes=Windows;WinProcs=Windows
```

Ceci assure la compatibilité ascendante, puisque les unités Windows et WinProcs ont été remplacées par l'unité Windows. Vous pouvez également ajouter là vos propres alias.

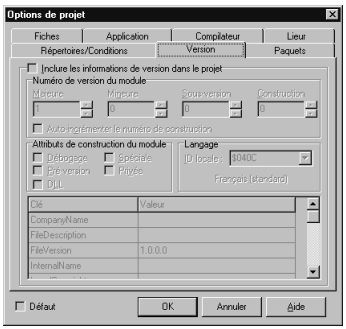
Astuce

Maintenant que nous avons vu ensemble les différentes options permettant de gérer votre projet, regardons les menus Fichier, Voir, Groupes de travail et Outils, ainsi que diverses options de la barre d'icônes. Vous allez découvrir quelques raccourcis bien pratiques.

Onglet Version

L'onglet Version, visible Figure 5.11, est utilisé pour associer à votre projet des informations de version, notamment des informations de copyright. Pour en savoir plus à ce sujet, reportez-vous à l'aide en ligne de Delphi.

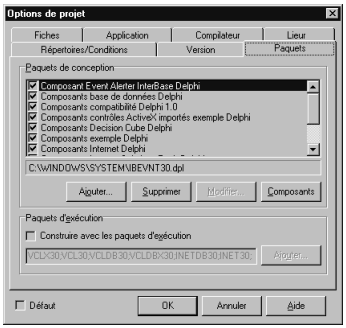
Figure 5.11
L'onglet Version.



L'onglet Paquets

L'onglet Paquets vous permet de spécifier les options de paquets de Delphi. C'est ici que vous sélectionnerez les paquets requis par votre projet et que vous activerez leur gestion. Les paquets sont des DLL particulières qui vous permettront de diminuer de façon significative la taille de vos exécutables.

Figure 5.12
L'onglet Paquets.

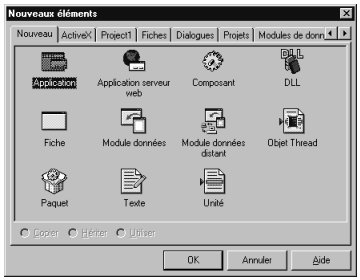


Créer un projet

La fin de cette journée est consacrée aux fonctions de gestion de projet de Delphi. Je vous recommande de compléter la lecture de cette leçon en consultant l'aide en ligne ainsi que les manuels Delphi. Si vous avez déjà utilisé Delphi 1.0, vous remarquerez l'apparition d'un certain nombre de nouveaux éléments, quelques modifications de noms et certaines modifications dans l'emplacement des menus. Lorsque vous démarrez Delphi, vous pouvez généralement voir une fiche et un projet vierges. La fiche et le projet vierges sont créés par défaut par Delphi. Vous pouvez modifier cette configuration par défaut, comme nous l'évoquerons rapidement. Vous connaissez maintenant la fiche et le code par défaut, intéressons-nous donc aux autres

options dont vous disposez lors de la création d'un projet. Dans le menu Fichier se trouve un menu appelé menu Nouveaux éléments. Vous pouvez trouver là toute une série d'onglets très pratiques (voir Figure 5.13).

Figure 5.13
La boîte de dialogue Nouveaux éléments.



Faisons un rapide tour d'horizon de ce qui est disponible dans chacun de ces onglets. Si vous souhaitez plus de détails sur tel ou tel élément, reportez-vous à l'aide en ligne ou aux manuels Delphi.

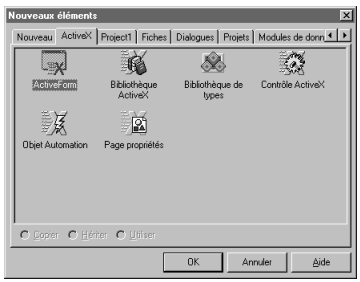
Onglet Nouveau

L'onglet Nouveau (voir la Figure 5.13) permet de créer de nouvelles applications, fiches, objets d'automatisation, objets d'édition de texte, composants et DLL. Cliquez sur l'icône qui vous intéresse et appuyez sur OK. Si Delphi a besoin d'informations supplémentaires concernant l'objet sélectionné, une nouvelle boîte de dialogue fera son apparition.

Onglet ActiveX

L'onglet ActiveX (voir Figure 5.14) vous permet de créer des contrôles et applications ActiveX. Cliquez sur l'icône qui vous intéresse et appuyez sur OK. Si Delphi a besoin d'informations supplémentaires concernant l'objet sélectionné, une nouvelle boîte de dialogue fera son apparition.

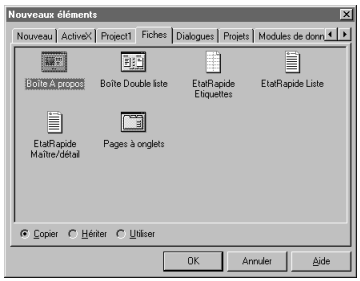
Figure 5.14
L'onglet ActiveX.



Onglet Fiches

L'onglet Fiches (voir Figure 5.15) vous propose divers modèles de fiches que vous pouvez utiliser lors de la création d'une nouvelle fiche. Vous pouvez également créer vos propres modèles et les faire figurer dans cet onglet, comme nous le verrons plus loin dans cette leçon. Cliquez sur l'icône qui vous intéresse et appuyez sur OK. Si Delphi a besoin d'informations supplémentaires concernant l'objet sélectionné, une nouvelle boîte de dialogue fera son apparition.

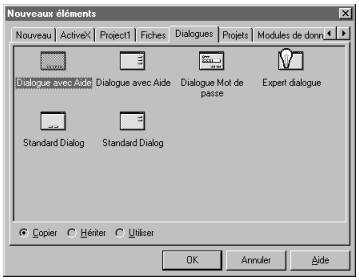
Figure 5.15
L'onglet Fiches.



Onglet Dialogues

Tout comme l'onglet Fiches, l'onglet Dialogues (voir Figure 5.16) contient des modèles pour des boîtes de dialogue préconstruites. Vous pouvez également créer de tels modèles et les faire figurer dans cet onglet. Cliquez sur l'icône qui vous intéresse et appuyez sur OK. Si Delphi a besoin d'informations supplémentaires concernant l'objet sélectionné, une nouvelle boîte de dialogue fera son apparition.

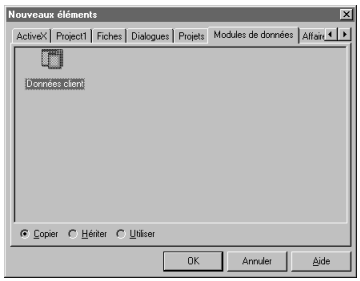
Figure 5.16
L'onglet Dialogues.



Modules de données

L'onglet Modules de données (voir Figure 5.17) permet de stocker des objets contenant des tables de données prédéfinies ainsi que d'autres composants. Un exemple de module clientèle est fourni, mais vous pouvez créer les vôtres en utilisant les informations données dans le Jour 11. Pour plus d'informations, consultez l'aide en ligne et les manuels Delphi. Cliquez sur l'icône qui vous intéresse et appuyez sur OK. Si Delphi a besoin d'informations supplémentaires concernant l'objet sélectionné, une nouvelle boîte de dialogue fera son apparition.

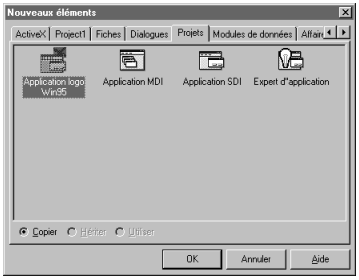
Figure 5.17
L'onglet Modules de données.



Onglet Projets

L'onglet Projets (voir Figure 5.18) contient des modèles de projets. Vous pouvez gagner du temps en utilisant les projets prédéfinis qui vous sont proposés, ou créer vos propres modèles. Cliquez sur l'icône qui vous intéresse et appuyez sur OK. Si Delphi a besoin d'informations supplémentaires concernant l'objet sélectionné, une nouvelle boîte de dialogue fera son apparition.

Figure 5.18
L'onglet Projets.

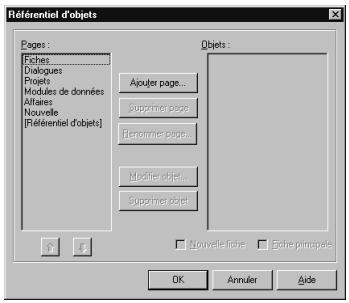


Un dernier point. Vous avez peut être déjà remarqué que si vous travaillez sur un projet, celui-ci sera ajouté au menu Nouveaux éléments, immédiatement après l'onglet Nouveau. Ceci vous permet de voir ce que vous avez ajouté, mais vous pouvez également y effectuer des sélections si vous devez créer une fiche ou un objet similaire à ceux qui figurent déjà dans le projet. Cet onglet disparaît dès que vous fermez le projet en cours.

Référentiel d'objets

En Delphi 1.0, la Galerie servait à stocker des modèles et des experts. Elle a été remplacée par le Référentiel. Le menu Options du référentiel d'objets se trouve dans le menu Outils/Référentiel que vous pouvez voir Figure 5.19.

Figure 5.19
La boîte de dialogue Référentiel d'objets.



Vous trouverez ici cinq choix : Experts fiches, Objets fiches, Experts projets, Modèles projets et Référentiel d'objets. Dans ce menu vous pouvez ajouter, supprimer et éditer des sélections, mais aussi modifier les paramètres par défaut utilisés lors de la création d'une nouvelle fiche ou d'un nouveau projet. Vous pouvez créer vos propres pages et objets, et les ajouter au référentiel à partir de cette boîte de dialogue. Pour en savoir plus à ce sujet, consultez l'aide en ligne.

Le référentiel d'objets est un excellent moyen de réutiliser du code dans Delphi. Comme vous pouvez le voir, les fonctions les plus usuelles sont déjà à votre disposition, mais vous pouvez étendre votre champ d'action bien au-delà en créant vos propres modèles que vous ajoutez au référentiel d'objets.

Nous allons maintenant parler de la création de modèles et d'experts. Cela nous permettra de nous faire une meilleure idée des processus en jeu.

Experts et modèles

Comme vous le savez sans doute, un expert est un programme installé avec Delphi qui vous fait passer par une série de questions et de choix, et qui, en s'appuyant sur vos réponses construit pour vous un cadre de travail permettant de bien commencer une fiche ou un projet complexe. Un expert peut vous faire gagner un temps précieux. Les modèles sont un peu similaires, en ceci qu'ils se chargent des aspects répétitifs à votre place, mais cette fois sans vous poser de questions permettant d'affiner le résultat. Avec un modèle, vous effectuez en fait une copie de ce que vous souhaitez, pour l'insérer dans votre nouveau projet. Les experts et les modèles apparaissent dans le menu Nouveaux éléments, et sont donc faciles d'accès.

Experts

Il existe deux types d'experts : les Experts de fiche et les Experts de projet. Pour savoir ce qui est disponible, vous pouvez aller dans le Référentiel des objets et cliquer sur l'onglet adéquat. Les experts disponibles sont affichés dans les onglets correspondant à leur fonction.

Pour utiliser un expert, il vous suffit de sélectionner Nouveau, puis de sélectionner l'expert souhaité dans le menu Nouveaux éléments. Une fois que l'expert est lancé, on vous demande diverses informations. Enfin, l'expert crée une fiche ou un projet et vous pouvez commencer à ajouter votre propre code, ou vous pouvez simplement y apporter quelques modifications.

Juste pour vous faire la main, sélectionnez Fichier/Nouveau. Puis, dans le menu Nouveaux éléments, sélectionnez l'onglet Dialogues. Double-cliquez sur l'icône Expert dialogue. L'expert dialogue démarre alors (voir Figure 5.20).

Figure 5.20
L'Expert dialogue.

Créer et utiliser des modèles

Un modèle est en fait un projet, une fiche ou une partie de code qui a été préconstruit pour être immédiatement utilisable. Lorsque vous sélectionnez un modèle, celui-ci est simplement copié dans votre projet, et vous pouvez y apporter les ajouts ou modifications que vous désirez. Par exemple, si vous avez créé une superbe fiche d'écran que vous tenez absolument à utiliser dans tous vos programmes, vous pouvez créer un modèle "superbe" contenant votre logo, vos couleurs, etc. Pour l'utiliser dans vos programmes, il vous suffit de l'enregistrer dans le Référentiel d'objets. Créez un modèle simple et utilisez-le dans un projet.

Si vous voulez créer votre propre modèle, concevez votre fiche ou votre application comme pour toute autre application ou fiche, puis, une fois la conception et les tests terminés, ajoutez-la au référentiel d'objets en procédant comme suit :

1. Sélectionnez Projet | Ajouter au référentiel. La boîte de dialogue Ajout au référentiel apparaît alors, comme sur la Figure 5.21.
2. Saisissez le titre et la description, sélectionnez la page dans laquelle vous désirez effectuer le stockage, saisissez le nom de l'auteur, et sélectionnez une icône.

Figure 5.21
La boîte de dialogue Ajouter au référentiel.

Appuyez sur OK. Voilà, vous venez de créer votre premier modèle. Pour utiliser un modèle, procédez comme suit :

1. Sélectionnez Fichier | Nouveau afin de faire apparaître la boîte de dialogue Nouveaux éléments.
2. Sélectionnez l'onglet dans lequel figure le modèle voulu.
3. Cliquez sur l'icône du modèle que vous désirez utiliser.

Chaque fois que vous lancez Delphi et que vous sélectionnez Fichier | Nouveau ou Fichier | Nouvelle application, vous avez recours à des modèles.

A mesure que vous vous habituez à la programmation Delphi et que vous commencez à bien cerner les tâches que vous souhaitez accomplir, il peut être utile de passer un peu de temps à créer des fiches et des boîtes de dialogue usuelles que vous convertirez en modèles. Plus votre collection de modèles sera étendue, plus vous gagnerez de temps sur les tâches de routine.

Contrôle de versions

Jusqu'ici, nous avons vu ce qui constitue un projet, comment s'organiser, comment travailler avec le Gestionnaire de projet de Delphi, et comment utiliser les fonctions permettant de gérer vos projets. Nous avons même parlé du gain de temps qu'offrent modèles et experts. Il est temps d'aborder le sujet du contrôle de versions. Etre bien organisé sans logiciel de contrôle de versions n'est pas très difficile si vous travaillez seul, mais peut le devenir si plusieurs programmeurs travaillent sur le même projet.

Un système de contrôle des versions vous permet de gérer des projets sans risque de désorganisation ou de pertes. Ainsi, si un groupe de programmeurs travaille sur le même projet, un système de contrôle de versions vous permet de verrouiller certaines parties du code que vous ne souhaitez pas voir modifiées, ou d'accéder à une partie du code et d'empêcher les autres membres du projet d'y apporter des modifications tant que vous travaillez dessus. Vous pouvez

conserver des archives des versions précédentes, ce qui vous permet de revenir en arrière ou de consulter une version plus ancienne de votre code.

Delphi prend en charge PVCS Version Manager 5.1 ou supérieur, logiciel qui est inclus dans l'édition Client/Serveur de Delphi. Les fonctions de contrôle de versions sont disponibles dans le menu Groupes. Pour accéder à ces fonctions, il vous suffit de suivre une procédure simple (pour plus de détails, reportez-vous à l'aide en ligne). Nous ne parlerons pas dans cet ouvrage du contrôle de versions, sachez simplement que ce produit existe et qu'il offre de nombreux avantages. Si vous développez seul une application en utilisant la version standard de Delphi, il vous suffit, pour arriver à un résultat similaire, d'adopter des méthodes simples de gestion de projet, en créant des répertoires distincts pour chaque version et en utilisant un ensemble standard de modèles. Par contre, si vous faites partie d'une équipe de développeurs travaillant sur un même projet, il est conseillé de vous munir du logiciel de contrôle de versions.

Récapitulatif

Lors de cette journée, nous avons vu tous les fichiers qui composent un projet Delphi, qu'ils soient créés au moment de la conception ou de la compilation. Nous avons parlé des fiches, des unités, de la VCL et des OCX en tant que composantes de votre projet, et nous avons vu comment inclure dans votre projet des fonctions, des procédures et des événements créés par l'utilisateur. Les ressources telles que les bitmap, les icônes, etc. n'ont pas non plus été oubliées. Nous avons ensuite vu comment mettre en place une structure de répertoires facilitant la recherche, l'entretien et la sauvegarde de vos fichiers. Avec le Gestionnaire de projet, vous avez appris comment naviguer dans votre projet, ajouter ou supprimer des fichiers. Nous avons vu la boîte à onglets des Options du projet, recelant diverses options permettant d'affecter la compilation de votre projet. Ensuite, nous avons parlé du Référentiel d'objets et vous avez appris à l'utiliser pour choisir comme défauts divers experts et modèles. Vous avez appris à vous servir d'experts pour créer fiches et projets. Nous avons vu comment choisir un modèle lors de la création d'un nouveau projet ou d'une nouvelle fiche, et comment créer vos propres modèles qui seront stockés dans le Référentiel des objets pour un usage ultérieur. Enfin, nous avons évoqué la prise en charge par Delphi du Gestionnaire de version PVCS d'Intersolv en soulignant les avantages de ce produit.

Atelier

L'atelier vous donne trois moyens de vérifier que vous avez correctement assimilé le contenu de cette leçon. La section Questions - Réponses reprend des questions posées couramment, et y répond, la section Questionnaire vous pose des questions, dont vous trouverez les réponses en consultant l'Annexe A, et les Exercices vous permettent de mettre en pratique ce que vous

venez d'apprendre. Tentez dans la mesure du possible de vous appliquer à chacune des trois sections avant de passer à la suite.

Questions - Réponses

Q Puis-je créer mes propres experts ?

R Oui, cela est théoriquement possible, et Delphi permet bien de créer facilement un expert. Les experts sont des produits ajoutés.

Q Pourquoi le système de contrôle de version PVCS ou son menu Groupes associés ne sont-ils pas disponibles dans le menu principal de Delphi ?

R Le système PVCS est un produit optionnel et doit être installé séparément. Pour l'activer, recherchez "PVCS" dans l'aide et suivez les instructions qui vous seront fournies.

Questionnaire

1. Quels sont les fichiers créés par Delphi lors de la conception ? Et lors de la compilation ?
2. Quelles sont les différentes parties qui composent une unité ?
3. Comment ajoutez-vous une fiche à un projet ? Comment supprimez-vous une fiche d'un projet ?
4. Si vous supprimez une fiche d'un projet, supprimez-vous le fichier de fiche associé ?

Exercice

Créez un modèle de projet contenant une fiche principale comportant des menus Fichier et Aide. Le menu d'aide devra comporter une option A propos. Ensuite, créez une fiche de boîte A propos... et faites figurer dans le bouton OK du code permettant d'enlever la fiche de la mémoire.

6

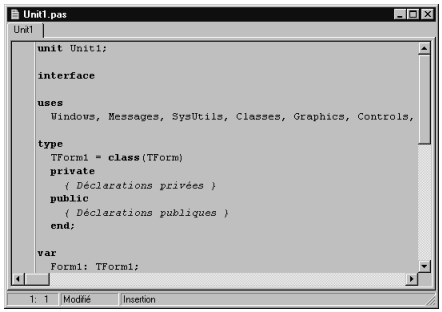
L'éditeur et le débogueur

Aux premiers temps de l'informatique, les programmeurs utilisaient nombre d'outils pour créer leurs logiciels. Parmi ces outils, on trouvait le plus souvent un éditeur, un compilateur, un éditeur de liens et un débogueur. Ces utilitaires étaient souvent fournis par différents éditeurs de logiciels, et rien ne les reliait, excepté la volonté du programmeur. Les premiers pionniers ne disposaient même pas de tous ces outils, ils en étaient réduits à relire des milliers de pages de code et à s'user les yeux sur des cartes perforées. De nos jours, les EDI (Environnements de Développement Intégrés) comprennent tous ces outils, regroupés de façon cohérente au sein d'une même interface afin de simplifier la vie du développeur. Cette leçon vous apprend à en tirer parti pour devenir plus productif. Nous commencerons par l'éditeur avant de nous pencher sur le débogueur intégré à Delphi.

L'éditeur

Comme vous le savez, l'éditeur de Delphi apparaît lorsque vous double-cliquez sur une fiche ou sur un contrôle, et vous amène à la section de code relative à l'objet sur lequel vous avez cliqué, comme indiqué Figure 6.1. De cette façon, Delphi bâtit le code source de l'application pour vous au fur et à mesure que vous ajoutez des composants et saisissez le code spécifique à votre application dans chaque section. Bien entendu, il vous est possible de parcourir le code et de rechercher pour révision des procédures déjà écrites.

Figure 6.1
L'éditeur de code.



Delphi met à votre disposition nombre de fonctionnalités qui vous simplifieront la vie. Mieux vous les connaîtrez, plus vite vous irez. En voici quelques-unes :

- Mise en évidence de la syntaxe,
- Annulations multiples,
- Edition Brief,
- Raccourcis clavier,
- Choix de l'organisation des raccourcis clavier,
- Menus d'accès rapide,

- Zone de messages d'erreurs,
- Mise en évidence des erreurs.

En outre, vous pouvez configurer les couleurs et le comportement général selon vos goûts. Pour vous familiariser avec l'éditeur et certaines de ces fonctionnalités, nous allons créer une petite application. Elle vous permettra d'étudier ensuite le fonctionnement du débogueur.

Fonctionnalités de l'éditeur et personnalisation

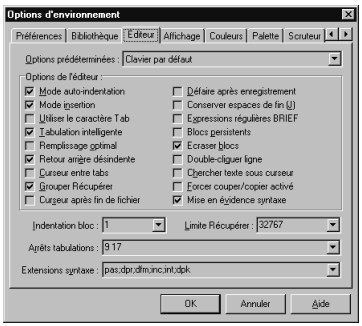
Avant de nous lancer dans la construction d'une application, nous allons détailler quelques options courantes des menus et des boîtes de dialogue utilisées pour personnaliser l'éditeur.

Sélectionnez Outils | Options d'environnement pour faire apparaître la boîte de dialogue Options d'environnement. Celle-ci vous propose plusieurs onglets, notamment Editeur, Affichage et Couleurs. Ces onglets comprennent de nombreuses options qui vous permettront de personnaliser l'éditeur. Passons-les en revue.

L'onglet Editeur

L'onglet Editeur (voir Figure 6.2) comporte plusieurs options de paramétrage du comportement de l'éditeur.

Figure 6.2
L'onglet Editeur.



Dans la zone de liste Options prédéterminées, vous pouvez d'un seul clic de souris modifier le paramétrage du clavier : Clavier par défaut, EDI classique, Emulation BRIEF, ou Emulation Epsilon. Les principales différences entre ces options sont les suivantes :

- Clavier par défaut : Auto-indentation, Insertion, Tabulation intelligente, Retour arrière désindente, Grouper récupérer, Ecraser blocs, et Mise en évidence de la syntaxe.
- IDE Classique : Auto-indentation, Insertion, Tabulation intelligente, Retour arrière désindente, Curseur entre tabs, Grouper récupérer, blocs persistents, et Mise en évidence de la syntaxe.

- Emulation BRIEF : Auto-indentation, Insertion, Tabulation intelligente, Retour arrière désindente, Curseur entre tabs, Curseur après fin de fichier, Conserver espaces de fin, Expressions régulières BRIEF, Forcer couper/copier activé, et Mise en évidence de la syntaxe.
- Emulation Epsilon : Auto-indentation, Insertion, Tabulation intelligente, Retour arrière désindente, Curseur entre tabs, Grouper récupérer, Ecraser blocs, et Mise en évidence de la syntaxe.

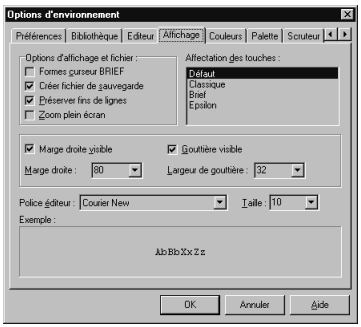
La zone Options de l'éditeur comporte plusieurs options que vous pouvez activer ou désactiver grâce à des cases à cocher. Ces paramétrages vous permettent un contrôle plus fin que les quatre options que nous venons de voir, qui sont simplement des combinaisons prédéfinies de ces options. Pour plus d'informations sur la fonction exacte de chaque option, consultez l'aide en ligne de Delphi.

Enfin, vous trouverez au bas de cette boîte de dialogue les options permettant de modifier l'indentation des blocs, le nombre d'annulations possibles (Limite Récupérer), les Arrêts de tabulations et les extensions de syntaxe.

L'onglet Affichage

L'onglet Affichage, repris Figure 6.3, est divisé en groupes qui vous permettent d'intervenir sur les options suivantes : Options d'affichage et de fichier, Affectation des touches, marges et polices.

Figure 6.3
L'onglet Affichage.



Les options d'affichage et de fichier vous permettent d'activer ou de désactiver les formes de curseur BRIEF, les fichiers de sauvegarde, la préservation des fins de ligne et le zoom plein écran.

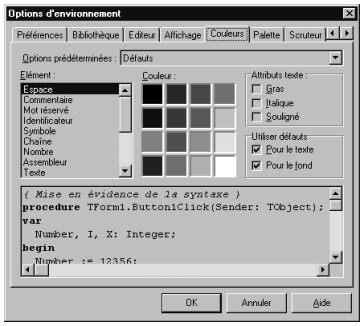
L'affectation des touches reprend les options que nous avons vues dans la section précédente (Défaut, Classique, Brief, ou Epsilon), mais les applique au clavier et non à l'éditeur. Par exemple, Ctrl+K+R lit un bloc depuis un fichier dans les modes Défaut et Classique, alors que le mode Brief utilise Alt+R.

Les options restantes vous permettent de configurer les marges, les gouttières et les polices, afin de vous donner un maximum de souplesse dans la gestion de l'apparence de la fenêtre d'édition.

L'onglet Couleurs

Cet onglet, repris Figure 6.4, offre plusieurs options permettant de personnaliser l'allure des différents éléments du code dans l'éditeur. C'est là que vous paramétrez la mise en évidence de la syntaxe. Comme dans les deux onglets précédents, des options prédéterminées sont à votre disposition. Vous pouvez, si vous le préférez, personnaliser vous-même chaque élément en utilisant le reste de l'onglet. La dernière zone vous permet d'observer le résultat de vos choix sans avoir à les appliquer à tout l'éditeur.

Figure 6.4
L'onglet Couleurs.



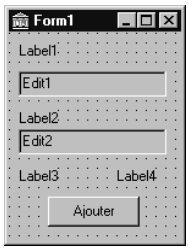
L'éditeur

Nous allons à présent créer une application des plus simples afin de faire connaissance avec l'éditeur. Cette application se contente de demander deux nombres à l'utilisateur, de les additionner lorsque celui-ci clique sur un bouton, puis d'afficher le résultat. La fiche se compose d'un bouton, de deux boîtes d'édition, et de quatre labels.

Si vous ne l'avez pas déjà fait, lancez Delphi. Vous voyez alors apparaître un projet vierge contenant une fiche par défaut. Si vous étiez en train de travailler sur une application, sélectionnez Fichier | Nouveau puis double-cliquez sur l'icône Application de la boîte de dialogue Nouveaux éléments.

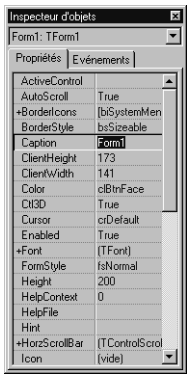
Une fois le nouveau projet créé, utilisez les étapes suivantes afin de construire l'application Addition. La Figure 6.5 vous guidera dans la disposition des composants sur la fiche.

Figure 6.5
L'application Addition.



1. Dans la palette des composants de Delphi, sélectionnez l'onglet Standard.
2. Dans la palette standard, cliquez sur l'icône représentant un bouton, puis cliquez sur la fiche. Vous verrez alors apparaître un bouton sur la fiche, bouton que vous pouvez redimensionner et déplacer à votre guise. Placez-le en bas de la fiche.
3. Pour donner un sens au bouton, modifions son intitulé. Cliquez sur le bouton afin de le sélectionnez, puis utilisez l'inspecteur d'objets pour donner à la propriété `Caption` du bouton la valeur `Ajouter`.

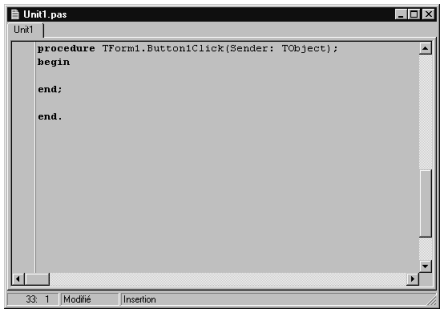
Figure 6.6
L'Inspecteur d'objets.



4. A présent, placez deux labels et deux boîtes d'édition sur la fiche, comme vous l'avez fait pour le bouton. L'icône de label ressemble à un A, et celle de la boîte d'édition... à une boîte d'édition. Placez de bas en haut `Label1`, `Edit1`, `Label2` et `Edit2`.
5. Tout comme pour le bouton, utilisez l'Inspecteur d'objets pour modifier l'intitulé de chacun des labels. Les propriétés `Caption` de `Label1` et `Label2` vaudront respectivement `Valeur1` et `Valeur2`.
6. Modifiez la propriété `Text` des deux boîtes d'édition en supprimant le texte qu'elles contiennent.

7. Ajoutez deux labels sous `Edit2`. Ces deux labels doivent être côte à côte, avec `Label3` à gauche et `Label4` à droite. Modifiez la propriété `Caption` de `Label3` et donnez-lui la valeur `Somme`. Effacez la propriété `Caption` de `Label4`.
8. Avant d'ajouter du code pour le bouton, vous devez appeler l'éditeur de code. Delphi créera automatiquement le code pour l'événement `OnClick` et le fera apparaître dans l'éditeur, comme indiqué Figure 6.7.

Figure 6.7
La fenêtre d'édition de code prête à recevoir le code de l'événement OnClick *du bouton.*



9. Vous allez à présent insérer le code nécessaire au bouton (`Button1`). Il devra ressembler à celui-ci :

```
procedure TForm1.Button1Click(Sender: TObject);
Var
   Valeur1,Valeur2,Total : integer;
begin
     Valeur1:=StrToInt(Edit1.Text);
     Valeur2:=StrToInt(Edit2.Text);
     Label4.Caption:=IntToStr(Valeur1+Valeur2);
end;
```

10. Pour enregistrer le projet, sélectionnez Fichier | Enregistrer sous. Sauvegardez l'unité sous le nom `FICHEADDITION.DPR`, et le projet sous le nom `ADDITION.DPR`.
11. Vous pouvez à présent tester l'application pour vous assurer qu'elle fonctionne. Pressez la touche F9 pour exécuter l'application. Si tout s'est bien passé, vous devez voir apparaître la fiche, dans laquelle vous saisissez deux nombres. Lorsque vous cliquez sur le bouton Ajouter, la somme apparaît normalement dans `Label4`.

Delphi ajoute du code à votre projet au fur et à mesure que vous ajoutez des composants sur la fiche. Dans l'étape 9, vous avez ajouté du code pour `Button1`. Le code définitif devrait ressembler à celui du Listing 6.1

Listing 6.1 : Le programme Addition

```
unit ADDITION;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Label1: TLabel;
    Label2: TLabel;
    Edit1: TEdit;
    Edit2: TEdit;
    Label3: TLabel;
    Label4: TLabel;
    procedure Button1Click(Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
var
   Valeur1,Valeur2,Total : integer;
begin
     Valeur1:=StrToInt(Edit1.Text);
```

```
    Valeur2:=StrToInt(Edit2.Text);
    Label4.Caption:=IntToStr(Valeur1+Valeur2);
end;

end.
```

Le code qui permet à cette application de fonctionner est celui que vous avez placé dans l'événement `OnClick` du bouton. Il se contente de prendre le texte saisi par l'utilisateur dans les boîtes d'édition, de le convertir du format chaîne au format nombre grâce à la fonction `StrToInt()`, de stocker les deux entiers dans deux variables nommées `Valeur1` et `Valeur2`, d'ajouter ces deux valeurs, de convertir le résultat au format chaîne, puis de placer le résultat dans la propriété `Caption` de `Label4`. Delphi a créé tout le reste du code.

Note

Si l'utilisateur saisit des données non numériques, une erreur se produit.

Lorsque vous utilisez l'éditeur, vous pouvez tirer profit des fonctionnalités vues précédemment, afin de le paramétrer selon vos goûts. Pour en savoir plus à ce sujet, reportez-vous à l'aide en ligne.

Nous avons passé en revue les principales options de l'éditeur, et vu comment utiliser l'éditeur pour saisir du code. Référez-vous à l'aide en ligne de Delphi ou aux manuels papier pour en savoir plus sur l'éditeur. Nous nous sommes contentés d'un aperçu de ces fonctionnalités, mais vous avez déjà pu apprécier la richesse de cet environnement de travail ; il dispose également bien évidemment de toutes les fonctions classiques de copier/coller que vous avez pu découvrir dans d'autres applications Windows. Nous allons à présent nous pencher sur le code.

Auditeur de code

Cette nouvelle fonctionnalité devrait intéresser aussi bien les novices que les programmeurs confirmés en Delphi. Si vous apprenez Delphi, il vous arrivera souvent de ne pas vous souvenir de la syntaxe d'une instruction, par exemple `If`, `For` ou `While`. Ce problème peut également survenir si vous passez souvent d'un langage à un autre : vous savez ce que vous voulez faire mais la syntaxe exacte vous échappe. Cette fonctionnalité vous fera gagner du temps et des frappes clavier, vous évitant nombre d'erreurs.

Voici les trois éléments qui peuvent vous aider :

- Expert Modèle de code
- Expert Achèvement de code
- Expert Paramètres de code

Supposons que vous soyez en train de taper du code et que vous désiriez commencer une instruction `If`. Tapez `If` puis Ctrl+J, et un menu surgissant apparaît, mettant à votre disposition toutes les variantes possibles de l'instruction `If`. Il vous suffit de sélectionner la variante désirée pour qu'elle soit automatiquement insérée dans le code. Il ne vous reste plus alors qu'à rem-

plir les blancs. Cette opération est réalisée grâce à un modèle de code. Delphi est fourni avec plusieurs modèles de code courant prêts à l'emploi.

L'expert Achèvement de code achève la ligne de code à votre place lorsque vous saisissez un nom de classe suivi d'un point (`.`). Ceci peut vous aider lorsque vous écrivez des composants complexes ou d'autres applications où certaines tâches sont fréquemment réutilisées. Pour avoir un exemple de cette fonctionnalité, essayez la classe `TApplication`. Tapez `Application` dans la fenêtre d'édition, et vous obtenez une liste des propriétés, méthodes et événements disponibles. Sélectionnez l'entrée qui vous intéresse et le code sera automatiquement saisi à votre place.

L'expert Paramètres de code affiche les paramètres des fonctions, procédures et méthodes sous forme de bulles d'aide. Il est désormais inutile de fouiner dans les fichiers d'aide à la recherche des paramètres exacts d'une fonction : tapez simplement le nom de la procédure, méthode ou fonction, ouvrez la parenthèse, et une bulle d'aide apparaît automatiquement pour vous rappeller les paramètres à utiliser. Par exemple, imaginons que vous tapiez :

```
FileOpen(
```

La bulle d'aide apparaît automatiquement dès que vous entrez la ligne précédente. Il ne vous reste plus qu'à continuer votre travail.

Vous êtes maintenant prêt à passer au débogage et à ses techniques. Nous reviendrons ensuite sur notre programme pour y insérer une erreur qui nous donnera l'occasion d'utiliser le débogueur.

Débogage

Quelle que soit l'application à programmer, vous commettrez toujours des erreurs.

Ce n'est pas parce que le code de votre projet se compile sans erreurs que votre application en est exempte. Certaines erreurs sont simples à retrouver et à corriger, d'autres peuvent vous demander de longues heures de travail. La première règle en la matière est de se souvenir qu'un utilisateur ne pense pas de la même façon qu'un développeur, et qu'en conséquence, il trouvera toujours dans votre programme des failles inattendues.

Comment limiter ces erreurs ? Commencez par anticiper le comportement de l'utilisateur en prévoyant la gestion des fichiers absents, des formats de données incorrects, etc. La liste des points susceptibles de poser problème est malheureusement longue, et elle dépend de ce que fait votre application.

Vous pouvez vous éviter nombre d'ennuis en gérant les erreurs et/ou en prévenant l'utilisateur du type de problème et des méthodes à suivre pour le corriger, quand c'est possible. Ne soyez pas surpris de voir un utilisateur planter votre application en quelques minutes alors que vous avez passé des heures, voire des semaines, à la rendre robuste. Quelqu'un trouvera toujours une façon de la planter. Lorsque c'est possible, demandez à d'autres personnes de tester votre appli-

cation. Demandez-leur de tester toutes les fonctionnalités et de vous informer des problèmes qu'ils rencontrent. De cette façon, vous pourrez corriger et prévenir les erreurs.

Lorsque vous découvrez une erreur, il vous faut localiser rapidement la portion de code où elle se produit. Vous trouverez parfois du premier coup d'œil, mais ce ne sera pas toujours le cas. On rencontre fréquemment deux types d'erreurs : des erreurs d'exécution, et des erreurs de logique.

Les erreurs d'exécution sont celles que le compilateur a laissé passer mais qui se produisent lors de l'exécution d'une portion spécifique de votre code. Par exemple, si vous tentez d'ouvrir un fichier absent du disque sans vous être assuré de sa présence.

Les erreurs de logique sont des imperfections dans la conception du code. Un bon exemple est une routine qui entre dans une boucle infinie, attendant un événement qui ne se produit jamais. Le programme s'arrête alors. Ce problème peut être causé par une simple faute de frappe.

Il existe bon nombre de scénarios pour chacune de ces catégories, et il est important de toujours les garder en tête lorsque vous écrivez du code ou lorsque vous le déboguez. Dans la section suivante, vous apprendrez à retrouver rapidement les erreurs.

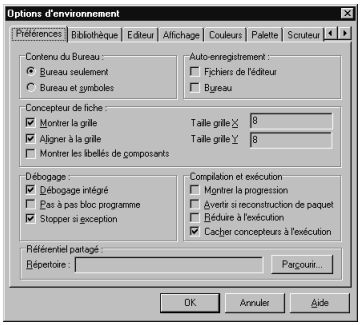
Le débogueur

Qu'est-ce qu'un débogueur ? Pour dire les choses simplement, c'est un logiciel qui vous permet d'exécuter votre code ligne par ligne afin d'examiner et de modifier la valeur des variables, constantes, objets, etc. Sans débogueur, vous êtes obligé d'insérer du code de Débogage afin d'afficher à l'écran, sur l'imprimante ou dans un fichier, la valeur des variables et autres informations que vous voulez surveiller. Ce processus prend du temps, et peut même masquer certaines erreurs, comme celles liées à un timeout. Delphi dispose fort heureusement d'un débogueur de belle facture intégré à l'EDI, qui vous permet de retrouver rapidement les erreurs afin de les corriger. Si vous n'êtes pas habitué aux débogueurs, il vous faudra un peu de temps et de pratique avant d'utiliser correctement cet outil, mais le jeu en vaut la chandelle. Commençons par un tour d'horizon du débogueur.

Paramétrage des options de débogage

Commençons par examiner les options de débogage de l'onglet Préférences de la boîte de dialogue Options d'environnement. Nous nous intéresserons dans un premier temps aux options de débogage suivantes : Débogage intégré, Pas à pas bloc programme et Stopper si exception.

Figure 6.8
L'onglet Préférences de la boîte de dialogue Options d'environnement.

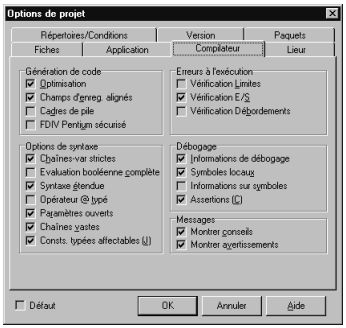


Le mieux est de conserver le paramétrage par défaut, mais voici une brève description de ces options :

- Débogage intégré : Active le débogueur intégré.
- Pas à pas bloc programme : Arrête le programme à la première initialisation d'une unité comportant du code de débogage (ce code est inséré par Delphi lors de la phase de compilation).
- Stopper si exception : Force le débogueur à s'arrêter pour afficher des informations relatives à l'erreur.

L'onglet Compilateur de la boîte de dialogue Options de projet (voir Figure 6.9) contient d'autres paramétrages relatifs au débogage. Pour accéder à cette boîte de dialogue, Sélectionnez Projet | Options. Le groupe de débogage comporte quatre paramètres : Informations de débogage, Symboles locaux, Informations sur symboles, et Assertions.

Figure 6.9
L'onglet Compilateur de la boîte de dialogue Options de projet.



Ici encore, nous vous conseillons de laisser les valeurs par défaut. Voici cependant la signification des options :

- Informations de débogage : Stocke les informations de débogage dans les fichiers `.DCU` lors de la compilation.

- Symboles locaux : Stocke des informations sur les symboles dans les fichiers `.DCU`, ce qui vous permet d'utiliser le débogueur intégré, ou un débogueur externe, afin d'examiner ou de modifier les variables locales au module.
- Informations sur symboles : Permet de générer des informations relatives aux objets de l'application, informations stockées dans les fichiers `.DCU`.
- Assertions : Permet de générer des assertions dans un fichier source. Cette option est activée par défaut, mais vous pouvez utiliser les directives de compilation `$C+` et `$C-` pour l'activer ou la désactiver dans le code.

Enfin, l'onglet Lieur de la boîte de dialogue Options de projet comporte une autre entrée intéressante : Inclure infos de débogage TD 32. Cette option doit être active si vous envisagez d'utiliser Turbo Debugger pour Windows. Quand elle est activée, cette option place des informations supplémentaires dans l'exécutable. Cette option est désactivée par défaut, et nous n'en aurons pas besoin dans le cadre de cet ouvrage.

Les points d'arrêt

Nous allons, dans l'application Addition créée précédemment, insérer quelques modifications (dont l'une provoquera des erreurs) afin de voir comment fonctionne le débogueur lorsque vous recherchez une erreur dans une application.

Tout d'abord, nous allons déplacer le code responsable de l'addition dans une procédure, et demander au gestionnaire d'événement `OnClick` d'appeler cette procédure. Pour ce faire, ajoutez la procédure suivante dans la section `implementation` :

```
implementation

{$R *.DFM}

procedure TForm1.AjouteNombre;
Var
   Valeur1,Valeur2,Total : integer;
begin
     Valeur1:=StrToInt(Edit1.Text);
     Valeur1:=StrToInt(Edit2.Text);
     Label4.Caption:=IntToStr(Valeur1+Valeur2);
end;
```

A présent, ajoutez la ligne suivante à l'événement `OnClick`, et supprimez le code qui y figurait précédemment :

```
procedure TForm1.Button1Click(Sender: TObject);
begin
     AjouteNombre;
end;
```

Enregistrez les fichiers sous ces nouveaux noms : `AdditionFiche2.pas` et `Addition2.dpr`.

Votre programme modifié devrait ressembler à celui du Listing 6.2. Assurez-vous qu'il est identique, car une erreur a été insérée.

Listing 6.2 : Application Addition contenant une erreur

```
unit ajoutefiche2;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs,  StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
    Edit2: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    procedure Button1Click(Sender: TObject);
  private
    { Déclarations privées }
    procedure AjouteNombre;
  public
    { Déclarations publiques }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.AjouteNombre;
Var
   Valeur1,Valeur2,Total : integer;
```

```
begin
     Valeur1:=StrToInt(Edit1.Text);
     Valeur1:=StrToInt(Edit2.Text);
     Label4.Caption:=IntToStr(Valeur1+Valeur2);
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
     AjouteNombre;
end;

end.
```

Lancez le programme et observez le résultat. Saisissez deux petits nombres à additionner et cliquez sur le bouton Ajouter. Le résultat devrait être incorrect. Utilisez le débogueur afin d'examiner le code et d'y trouver la cause du problème.

Le première chose à faire est d'établir un point d'arrêt. Un point d'arrêt force le programme à s'arrêter à la ligne sur laquelle il est placé, sans exécuter ladite ligne.

Pour définir un point d'arrêt, cliquez dans la gouttière (la zone grisée à gauche du code dans la fenêtre d'édition) de la ligne sur laquelle vous voulez définir un point d'arrêt ; dans notre cas, la ligne suivante :

```
Valeur1:=StrToInt(Edit1.Text);
```

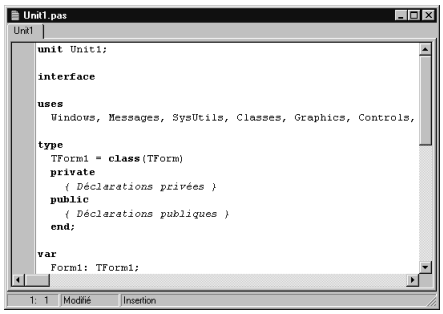
La ligne devrait apparaître en surbrillance rouge, avec un point rouge situé dans la gouttière, comme indiqué Figure 6.10 (en supposant que vous utilisiez le paramétrage par défaut de Delphi).

Note

La gouttière est visible par défaut, mais elle peut être désactivée par les options d'environnement (onglet Affichage, en décochant la case gouttière). Si la gouttière est invisible, cliquez sur le premier caractère de la ligne où vous désirez définir le point d'arrêt, et la ligne apparaîtra en rouge.

Si vous définissez un point d'arrêt sur une ligne de code optimisée par Delphi, vous récupérerez au moment de la compilation une boîte de dialogue d'avertissement vous informant de ce qui s'est produit et vous demandant si vous désirez quand même continuer. La même boîte de dialogue apparaît quand vous tentez de définir un point d'arrêt sur une ligne de déclaration de variables. Lorsqu'une variable que vous désirez examiner a été optimisée de la sorte, choisissez Voir | Gestionnaire de projet, et sélectionnez l'onglet Compilation dans la boîte de dialogue Options de projet. Vous pourrez alors désactiver l'optimisation dans la section Génération de code. Reportez-vous à l'aide en ligne pour avoir plus de détails.

Figure 6.10
L'éditeur de code, gouttière visible.

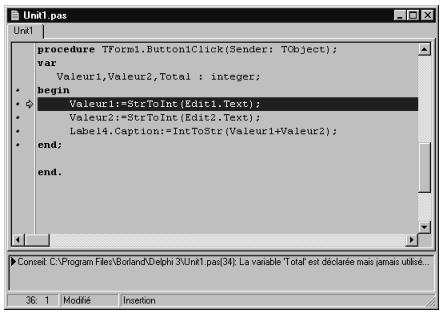


Note

La gouttière est une innovation de Delphi 3. Elle permet de repérer visuellement les lignes de code, les lignes comportant un point d'arrêt et la ligne en cours d'exécution. Lorsqu'elle est invisible, ces lignes se repèrent en fonction de leur couleur dans l'éditeur de code.

Exécutez le programme à nouveau. Tapez 4 comme valeur 1, et 5 comme valeur 2, puis appuyez sur le bouton Ajouter. L'éditeur de code devrait apparaître, indiquant l'emplacement du point d'arrêt, comme indiqué Figure 6.11. Le point d'arrêt fait à présent apparaître une coche verte dans la gouttière, indiquant que le programme est arrêté à cet endroit. La ligne n'a donc pas encore été exécutée. En d'autres termes, la coche verte désigne la prochaine ligne de code qui sera exécutée.

Figure 6.11
L'éditeur de code en mode débogage, indiquant l'endroit où s'est arrêté le programme.



Une dernière chose concernant les points d'arrêt : vous pouvez définir des points d'arrêt spéciaux qui ne stopperont l'exécution du code que lorsqu'une certaine condition sera remplie, ou lorsque le passage à la ligne mentionnée se sera fait un certain nombre de fois. Ceci peut être utile, notamment lorsque votre code échoue au milieu d'une très longue boucle. Pour utiliser ces fonctionnalités avancées, utilisez l'option de menu Exécuter | Ajouter point d'arrêt afin d'accéder à la boîte de dialogue Modification de point d'arrêt. Nous n'en parlerons pas plus longuement ici ; référez-vous à l'aide en ligne pour plus d'informations.

Autres options de débogage

Le menu Exécuter de Delphi comporte d'autres options de débogage, avec leur raccourci clavier :

- Exécuter (F9) : Compile et exécute, ou (en mode débogage) reprend l'exécution.
- Pas à pas (F8) : Exécute le code ligne par ligne, sans tracer dans les appels de fonctions et procédures.
- Pas à pas approfondi (F7) : Exécute le code pas à pas, y compris les procédures et fonctions.
- usqu'à la prochaine ligne (Shift+F7) : Va à la ligne suivante.
- Jusqu'au curseur (F4) : Continue l'exécution jusqu'au curseur.
- Montrer le point d'exécution : Place le curseur sur le point d'exécution.

A présent que vous maîtrisez les options de débogage, revenons à notre programme d'addition.

Examiner la valeur des variables en utilisant des points de suivi

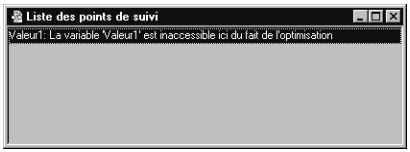
Il existe deux façons d'examiner la valeur stockée dans une variable : en utilisant des points de suivi, ou alors une nouvelle fonctionnalité appelée l'évaluateur d'expressions (dont nous parlerons dans la prochaine section).

Un point de suivi vous permet d'indiquer le nom des variables que vous désirez surveiller et d'avoir leur valeur affichée dans une petite fenêtre. Pour ajouter un point de suivi, vous pouvez soit appuyer sur Ctrl+F5, soit sélectionner Exécuter | Ajouter point du suivi. Vous accéderez ainsi à la boîte de dialogue Propriétés du point de suivi présentée Figure 6.12. Vous pouvez pour chaque point de suivi spécifier le nom de variable, le type, et des expressions telles que `Valeur1+Valeur2` ou `IntToStr(Valeur1)`.

Figure 6.12
La boîte de dialogue Propriétés du point de suivi.

Dans cette boîte de dialogue, tapez `Valeur1` dans la zone de texte Expression. Vous devriez alors voir apparaître la fenêtre de suivi dans laquelle les variables suivies ainsi que leur valeur s'affichent, comme indiqué Figure 6.13.

Figure 6.13
La boîte de dialogue Liste des points de suivi.



Si la variable n'a pas encore été créée ou n'a pas encore reçu d'affectation, différents messages sont affichés, indiquant le statut de la variable, à la place de sa valeur. Le message peut être Valeur inaccessible, ce qui signifie soit que le code qui crée la variable n'a pas encore été exécuté, soit qu'il a été exécuté et que la variable a été retirée de la mémoire. Vous pouvez également voir apparaître le message Inaccessible ici du fait de l'optimisation, comme indiqué Figure 6.13, ce qui signifie le plus souvent que la variable n'a pas encore été affectée. Lorsque les lignes affectant ou modifiant la valeur de la variable sont exécutées, cette valeur apparaît dans la fenêtre.

Dans notre application Addition, ajoutez un point de suivi pour les variables `Valeur2` et `Total`. Pour ce faire, accéder à la boîte de dialogue de suivi d'expression comme indiqué ci-avant, tapez le nom de la variable que vous désirez surveiller (par exemple `Valeur2`) dans la zone de texte Expression, puis cliquez sur OK. Vous devez procéder de même pour chacune des variables que vous désirez surveiller. Appuyez ensuite sur F8 (Pas à pas) deux fois afin de passer dans le reste de la procédure, ce qui vous permettra d'inspecter le contenu des variables. Que découvrez-vous alors ? `Valeur1` a la valeur qui aurait dû être affectée à `Valeur2`, `Valeur2` comporte une valeur étrange que vous ne lui avez jamais affectée, et `Total` comporte un message indiquant qu'il a été supprimé par le lieur.

Revenez à la procédure `AjouteNombre` du Listing 6.2, et vous verrez que la valeur de `Edit2.Text` a été affectée à `Valeur1` au lieu de `Valeur2`. Cette erreur se produit le plus souvent lorsqu'en recopiant une ligne par un copier/coller, vous oubliez d'adapter la ligne après sa copie. De ce fait, `Valeur2` n'a jamais été initialisée, et contient une valeur aberrante.

Voici pour `Valeur1` et `Valeur2`, mais que s'est-il passé pour `Total` ? Dans notre exemple, cette variable n'a jamais été utilisée, le lieur l'a donc supprimée. Ceci se produit souvent lorsque vous laissez dans votre code des variables que vous pensiez utiles et qui n'ont jamais servi. Il ne s'agit pas à proprement parler d'une erreur, mais le compilateur vous signale le fait afin que vous puissiez, si vous le désirez, supprimer la variable incriminée.

Indicateur de statut pendant le débogage

Remarquez la zone de liste située sous l'éditeur. Elle affiche les erreurs, les avertissements, etc. Des barres de défilement apparaissent quand tous les messages ne tiennent pas dans la zone visible (cf. Figure 6.11). Vous trouverez là des informations utiles pendant les sessions de débogage.

Voici les erreurs affichées par le programme Addition :

```
Conseil:…(36): Valeur affectée à 'Valeur1' jamais utilisée.
Avertissement:…(38): La variable 'Valeur2' n'est peut-être pas initialisée.
Conseil:…(34): La variable 'Total' est déclarée mais jamais utilisée dans
'AjouteNombre'.
```

Maintenant que vous avez découvert certaines erreurs en utilisant le débogueur, vous pouvez modifier le code afin de les corriger et faire fonctionner correctement le code. Assurez-vous de sortir du programme avant d'effectuer des corrections. Pour ce faire, appuyez sur CTRL+F2. Revenez à la procédure `AjouteNombre` et changez `Valeur1` en `Valeur2` dans la deuxième ligne après le `begin`. Supprimez également la variable `Total` de la section de déclaration des variables. Relancez le programme en tapant F9. Lorsque vous appuierez sur le bouton après avoir saisi deux valeurs, le programme s'arrêtera au point d'arrêt que vous aviez défini, mais les erreurs auront disparu ; elles devraient apparaître conformes à vos attentes au fur et à mesure de votre parcours du code (touche F8).

Examiner la valeur des variables avec l'Evaluateur d'expressions

La deuxième façon d'inspecter la valeur des variables utilise une nouvelle fonctionnalité appelée l'Evaluateur d'expressions. Vous pouvez placer le curseur sur une variable, un objet ou un paramètre, et sa valeur s'affichera dans une info-bulle placée à côté du curseur.

Cette méthode est beaucoup plus rapide et simple d'emploi que l'utilisation des points de suivi. Vous gagnerez un temps précieux si vous ne vous intéressez qu'à un petit nombre de variables sur une petite section de code.

Déboguer une DLL

Delphi 3 dispose d'une nouvelle fonctionnalité de débogage qui vous évitera pas mal de nuits blanches. Il est à présent possible de déboguer les DLL Delphi en utilisant le débogueur de Delphi. En définissant un point d'arrêt dans le code d'une DLL, si vous fournissez le nom d'un programme appelant la DLL, Delphi lance l'application hôte et attend que cette dernière appel-

le la DLL. L'exécution sera suspendue dès que le point d'exécution atteindra un point d'arrêt dans la DLL. Voilà qui devrait vous simplifier la tâche.

Dans le passé, il fallait plusieurs programmes de débogage, ou bien extraire le code des DLL pour le faire tourner dans une application Delphi classique, en espérant que tout se passe bien une fois le code réintégré dans la DLL. Bien que ces méthodes aient fait leurs preuves, la méthode proposée par Delphi est beaucoup plus agréable et rapide d'emploi.

Sans trop entrer dans les détails, voyons cependant comment se passe une session de débogage de DLL en Delphi. Créez la DLL suivante en choisissant Fichier | Nouveau et en sélectionnant DLL dans la boîte de dialogue Nouveaux éléments, puis enregistrez le résultat sous le nom `MYMATH.DPR`.

Listing 6.3 : Débogage d'une DLL, listing d'exemple

```
library mymath;

{ Remarque importante à propos de la gestion mémoire des DLL: ShareMem
  doit être la première unité dans la classe USES de votre bibliothèque
  ET dans la classe USES de votre projet (sélectionnez Voir-Source du
  projet) si vos DLL exportent des procédures ou des fonctions passant en
  paramètre des chaînes ou des résultats de fonction. Ceci s'applique à
  toutes les chaînes passées par ou à vos DLL--mêmes celles qui sont
  imbriquées dans des enregistrements ou des classes. ShareMem est l'unité
  d'interface au gestionnaire de mémoire partagée DELPHIMM.DLL, qui doit
  être déployé avec vos propres DLL. Pour éviter l'emploi de DELPHIMM.DLL,
  passez vos chaînes en utilisant des paramètres PChar ou ShortString. }

uses
  SysUtils,
  Classes;
function Sqr(Num : integer):integer;export;
begin
     Sqr:=Num*Num;
end;
exports
    Sqr;
begin
end.
```

Analyse du code

Ce programme crée une DLL disposant d'une fonction appelée `sqr`*, qui se contente d'élever un nombre au carré avant de renvoyer le résultat au programme appelant.*

Il nous faut maintenant un programme à même d'appeler la DLL afin de la déboguer. Créez une nouvelle application Delphi en sélectionnant Fichier | Nouvelle application. Disposez sur cette fiche trois labels, un bouton et un champ d'édition, et utilisez le code du Listing 6.4. Enregistrez l'unité sous le nom `SQRTEST.PAS` et le projet sous le nom `TESTSQR.DPR` (dans le répertoire où vous avez créé la DLL). Sélectionnez Projet | Tout construire.

Listing 6.4 : Application TESTSQR servant à tester la DLL pour débogage

```
unit sqrtest;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;


type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Button1: TButton;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

function Sqr(Num : integer):integer;far;external 'mymath.dll';
implementation

{$R *.DFM}
```

```
procedure TForm1.Button1Click(Sender: TObject);
begin
     Label3.Caption:=IntToStr(Sqr(StrToInt(Edit1.Text)));
end;

end.
```

Info

Analyse du code

Ce code appelle la fonction `sqr` définie dans la DLL que vous avez créée au Listing 6.3. La valeur de `Edit1.Text` est passée à la DLL lors de l'appel à `sqr`. La valeur renvoyée par la DLL est affichée dans `Label3.Caption`.

Une fois ces deux programmes compilés sans erreur, testez la DLL en exécutant le programme `TESTSQR.EXE`. Si tout se passe bien, une fenêtre apparaît dans laquelle vous pouvez saisir une valeur (cf. Figure 6.14). Lorsque vous cliquez sur le bouton Carré, le carré du nombre devrait apparaître. Par exemple, tapez `4` et appuyez sur le bouton Carré. `16` s'affiche alors dans la boîte d'édition. A présent, éditez le source de la DLL et choisissez Exécuter | Paramètres pour obtenir la boîte de dialogue Paramètres d'exécution. Saisissez le chemin d'accès et le nom de l'application hôte, dans notre cas : `TESTSQR.EXE`. Lorsque vous exécutez, Delphi lance l'application hôte mais s'arrête à tout point d'arrêt défini dans la DLL. Les procédures de débogage sont alors identiques à celles des applications classiques.

Vous avez vu à quel point le débogage des programmes est facilité par le débogueur intégré à Delphi. Mais que faire lorsque quelque chose d'inattendu se produit au niveau de Windows ou dans des programmes que vous n'avez pas écrits ? Cette question nous amène à la dernière section de cette journée, consacrée à WinSight32.

WinSight32

Une étude approfondie de WinSight32 sortirait du cadre de ce livre, d'autant plus que cet outil repose sur le fonctionnement interne de Windows et que les connaissances associées sont difficiles d'accès. C'est pourquoi cette présentation de WinSight32 sera des plus brèves : il ne s'agit que de vous mettre au courant de l'existence de cet outil.

WinSight32 fait partie de la famille des espions système. Il permet d'examiner les process Windows en cours d'exécution, de tracer les fenêtres et process enfant, d'intercepter les messages système, et plus encore. Si vous rencontrez des problèmes avec une application (le plus souvent des problèmes de communication entre Windows et votre application ou une autre), WinSight32 peut vous permettre de vérifier les messages passant entre les deux process. Reportez-vous à la documentation en ligne si vous désirez approfondir le sujet.

Récapitulatif

Lors de cette journée, vous avez découvert des fonctionnalités et des options de l'éditeur de Delphi, notamment des systèmes de complétion automatique de commande ou de rappel des paramètres sous forme d'info-bulles. Vous avez appris à utiliser le débogueur, aussi bien dans des applications Delphi classiques que dans des DLL. Enfin, nous avons très brièvement parlé de WinSight32, un outil qui vous permet d'observer les objets, messages et process Windows.

Nous ne vous avons donné dans cette partie qu'un aperçu des techniques de débogage, mais vous devriez cependant en savoir assez pour savoir quelles informations rechercher et quelles méthodes adopter lors de votre prochaine session de débogage. N'hésitez pas à vous reporter à l'aide en ligne de Delphi ainsi qu'aux manuels pour en savoir plus.

Atelier

L'atelier vous donne trois moyens de vérifier que vous avez correctement assimilé le contenu de cette leçon. La section Questions - Réponses reprend des questions posées fréquemment et y répond, la section Questionnaire vous pose des questions dont vous trouverez les réponses en consultant l'Annexe A, et les Exercices vous proposent de mettre en pratique ce que vous venez d'apprendre. Tentez dans la mesure du possible de vous appliquer à chacune des trois sections avant de passer à la suite.

Questions - Réponses

Q Y a-t-il d'autres touches de raccourci actives dans l'éditeur ?

R Oui. Si vous avez déjà pratiqué des éditeurs WordStar, vous devriez être familier de Ctrl+Y pour effacer une ligne. Soyez cependant prudent : si vous passez en environnement Brief ou Epsilon, les raccourcis clavier avec lesquels vous avez grandi peuvent se révéler surprenants.

Q La case Application hôte est grisée dans la boîte de dialogue Paramètres d'exécution. Pourquoi ?

R Cette zone n'est utilisable que lorsque vous êtes en train de travailler sur une DLL.

Questionnaire

1. Quelles sont les options prédéterminées de l'éditeur ?
2. Où active-t-on la mise en évidence de la syntaxe ?
3. A quoi la gouttière sert-elle ?
4. Comment débogue-t-on une DLL ?
5. Qu'est-ce que WinSight32 ?
6. Comment peut-on évaluer la valeur des variables sans utiliser les points de suivi ?
7. Lorsque vous définissez un point d'arrêt sur une ligne, cette ligne est exécutée avant que le programme suspende son exécution. Vrai ou Faux ?
8. Comment relance-t-on l'exécution d'un programme après un point d'arrêt ?

Exercices

1. Dans le menu Outils | Options d'environnement, modifiez les options d'affichage et les options de l'éditeur. Parcourez l'aide en ligne afin de connaître les fonctionnalités et les raccourcis clavier des options que vous avez choisies. Effectuez cette démarche pour au moins deux options, dont l'option par défaut.
2. Lancez WinSight32, testez les différentes options et parcourez l'aide en ligne.

7

Concevoir une interface graphique

Une des grandes qualités de Windows pour un utilisateur c'est que la plupart des applications ont un aspect et un ordonnancement similaires. Une fois que vous avez utilisé plusieurs applications, vous pouvez presque automatiquement deviner où se trouvera telle ou telle fonction dans une application que vous utilisez pour la première fois. Ceci est valable pour tout, de l'installation au menu d'aide qui se trouvera toujours à l'extrême droite de la barre de menus. Vous avez déjà dû vous rendre compte de cela, sans savoir forcément pourquoi tout semble se conformer à quelque mystérieuse règle.

Microsoft a donné des spécifications de conception pour les logiciels Windows afin de modifier la philosophie de l'utilisateur. Le plus important n'est plus d'apprendre les nuances de l'interfaçage avec votre application, mais de devenir productif avec elle. Microsoft essaye de faire appliquer ces spécifications dans le cadre de son programme d'action Logo Win95. Selon ce nouveau programme, votre application doit satisfaire aux critères Microsoft si vous souhaitez pouvoir faire figure "Conçu pour Windows 95" sur l'emballage de votre logiciel. Lorsque vous voyez un tel logo sur un logiciel ou un périphérique, vous pouvez être sûr que le produit fonctionnera parfaitement dans l'environnement Windows 95 et qu'il est conçu en accord avec les spécifications de conception. Delphi 3 est conçu pour vous aider à développer des applications qui satisfont aux spécifications du Logo Win95. Borland vous fournit les outils, mais le plus gros des efforts vous incombe. Lors de cette journée, je vous montrerai comment concevoir votre application pour la rendre plus facile d'usage, et comment la rendre éligible pour le logo Windows 95.

Pourquoi une interface graphique utilisateur ?

Après tout, est-ce qu'une *interface graphique utilisateur* (GUI en anglais) est vraiment indispensable ? La réponse est non. Les utilisateurs ont utilisé des lignes de commandes pendant des décennies et ont survécu, et rien ne nous empêcherait d'en utiliser pendant des décennies encore. Sauf peut-être qu'on se rendrait terriblement malheureux. Quand des utilisateurs DOS passent à Windows 3.1 puis à Windows 95 leurs yeux s'illuminent à mesure qu'ils comprennent ce que ces interfaces graphiques peuvent signifier en terme de gain de productivité.

Nouveau

Une interface utilisateur graphique (IGU ou GUI) est un type de format d'affichage qui permet à un utilisateur de choisir des commandes, de lancer des programmes et d'afficher des listes de fichiers et d'options en pointant sur des représentations pictographiques (des icônes) et sur des listes d'éléments de menu présents à l'écran. On effectue généralement des choix au moyen d'un clavier ou d'une souris. Windows 95 est un exemple de GUI.

Les GUI existent parce que nous en avons besoin. Quand on doit chaque jour accéder à pas moins de 12 applications, on passerait son temps à lancer et à refermer des applications si on ne disposait que d'une ligne de commande et il resterait peu de temps pour vraiment travailler. Tout est question de productivité. La productivité s'améliore si l'on peut accéder simultanément à plusieurs applications. Basculer entre cc:Mail, Delphi 3, WinZip, WinCim, l'Explorateur, etc. est un jeu d'enfant dans l'environnement Windows. Inutile de refermer une application pour accéder à une autre, ce qui fait gagner un temps fou.

Prototypage et développement rapide

Nouveau

Vous avez peut-être déjà entendu ce terme barbare de prototypage rapide. Ce terme décrit la méthode consistant à créer des modèles fonctionnels de l'interface de votre application dans les premiers temps du processus de conception.

Vous pouvez alors présenter cette interface à vos clients et recueillir leurs impressions sur le comportement, le placement des objets, la façon dont les fonctions sont incluses, etc., tout en leur donnant le sentiment que vous vous préoccupez de leurs désirs. Plusieurs produits existent qui se chargent de faciliter le prototypage rapide. Ces logiciels construisent des applications en "trompel'œil" qui ne traitent aucune donnée mais donnent une bonne idée de l'interface. Les choses ont évolué cependant, et ces logiciels sont devenus obsolètes.

Avec l'arrivée d'une nouvelle génération d'outils RAD comme Delphi 3 et Visual Basic 4, ces outils de prototypage rapide ne sont plus nécessaires. Vous pouvez maintenant utiliser les outils RAD eux-mêmes pour créer ces squelettes d'application destinés à la démonstration.

Nouveau

Le terme de RAD (développement rapide d'applications) est assez parlant. A l'aide d'un ensemble d'outils sophistiqués (Delphi 3, par exemple), le programmeur développe rapidement des applications finalisées. En effet, les outils permettent au programmeur de mettre en œuvre les fonctionnalités les plus complexes sans fournir un travail démesuré. Ainsi, vous pouvez utiliser les boîtes de dialogue Ouvrir et Enregistrer, qui sont des boîtes de dialogues communes de Windows 95, au lieu de devoir les concevoir en partant de zéro. Vous pouvez aussi utiliser le contrôle conteneur OLE pour incorporer une feuille de calcul Excel dans l'application, sans devoir écrire un tableur en Delphi. Delphi 3 est l'exemple même d'un environnement de développement RAD, puisqu'il met à votre disposition des outils permettant de créer de très bonnes applications en peu de temps et moyennant peu de codage.

Voilà donc résolu le problème de la répétition des efforts. Quand vous créiez une interface de démonstration et que le client s'en estimait satisfait, vous deviez recoder l'interface en utilisant les outils permettant de générer une véritable application. Il fallait faire deux fois la même chose. Avec Delphi 3, vous pouvez commencer par développer l'interface, la montrer à votre client, puis terminer l'application, et tout cela dans le même environnement.

Pour que la conception de votre GUI se déroule bien, vous devez prendre un autre facteur en considération : votre équipe de développement de GUI. Vous travaillez peut-être seul pour l'instant, mais à mesure que les produits que vous développerez gagneront en taille et en complexité, vous aurez besoin d'aide. Lorsque vous mettez sur pied une équipe de développement GUI, n'allez surtout pas faire appel à des programmeurs ! Les programmeurs sont les pires concepteurs car ils estiment évidentes nombre de choses qui dépassent l'utilisateur moyen. Votre équipe de développement GUI doit être constituée de personnes venant d'horizons divers, parmi lesquelles des rédacteurs, ergonomistes, mais aussi des utilisateurs de divers niveaux de compétence. Evidemment vous ne disposez peut-être pas des ressources nécessaires à l'embauche de tout ce petit monde, mais vous comprenez le principe. La constitution d'une bonne équipe est primordiale au développement d'une interface utilisateur réellement efficace.

Une fois que votre produit a été codé, les tests d'utilisabilité vous montreront ce que vaut votre conception. Le meilleur testeur de votre logiciel est votre vieille maman qui ne connaît rien à l'informatique. Il est inutile de prendre en considération les exigences d'un crack de l'informatique en ce qui concerne l'interface utilisateur, à moins que votre logiciel ne soit justement destiné à un public de "cracks".

Priorité à l'utilisateur

Lorsque vous construisez votre application, il est important que vous gardiez à l'esprit les principes qui ont guidé l'écriture de Windows 95. Pour satisfaire les besoins de l'utilisateur, Microsoft s'appuie sur sept principes. Examinons-les ensemble.

- L'utilisateur a le contrôle
- Etre direct
- La cohérence
- Etre magnanime
- Du répondant
- De l'esthétique
- De la simplicité

L'utilisateur a le contrôle

Personne n'aime se sentir manipulé, à plus forte raison par un ordinateur. L'utilisateur doit toujours sentir qu'il a le contrôle de ce qui se déroule à l'écran. Il doit avoir l'impression que c'est lui qui prend l'initiative d'une action au lieu de réagir aux désirs de l'ordinateur. Si vous comptez automatiser grandement votre application, vous devez vous assurer que l'utilisateur garde le contrôle sur ces événements automatisés.

Chaque utilisateur est un individu à part. Il a des goûts et des besoins qui lui sont propres. Il est important de prévoir un moyen de personnaliser votre application. Voyez par exemple comme il est aisé de personnaliser l'interface de Windows 95. La possibilité de modifier les polices, les couleurs, les icônes et tant d'autres éléments permet de donner un aspect personnel au système. La plupart de ces attributs Windows vous sont accessibles à vous, le programmeur. Tirez-en partie et donnez la possibilité à votre programme de suivre les choix de couleurs et de polices déjà en place dans le système sinon, votre application semblera rigide et austère.

Avez-vous déjà regardé la façon dont une bonne application interagit avec l'utilisateur ? Une bonne application vous indique ce qui se passe ou dans quel "état" ou quel "mode" vous vous trouvez. Si vous êtes en mode écrasement dans Microsoft Word, les lettres RFP apparaissent dans la barre d'état. Votre application doit être aussi interactive que possible. Elle doit savoir répondre aux sollicitations et ne jamais laisser l'utilisateur se demander ce qui se passe.

Etre direct

Permettez aux utilisateurs de manipuler directement les objets dans leur environnement. La phrase "une image vaut mille mots" prend chaque jour plus de sens. Il est bien plus facile de se souvenir à quoi ressemble quelque chose que de mémoriser la syntaxe d'une commande. Je vois souvent des utilisateurs se souvenir comment accomplir une tâche en disant "Je fais glisser ce machin et je le pose sur ce truc-là et ça imprime". Même si ce n'est pas le cas pour vous, c'est comme ça que la plupart des gens raisonnent. Essayez de concevoir votre logiciel pour qu'il soit visuellement intuitif. Permettez aux utilisateurs de voir comment ce qu'ils font affecte les objets à l'écran.

Un des modes d'interaction les plus directs entre l'homme et l'ordinateur se fait par l'intermédiaire de métaphores. L'usage de métaphores a une bonne part dans le succès du Macintosh. La notion de dossier est bien plus parlante que celle de répertoire ou de fichier. Une "armoire" est remplie de "dossiers", qui contiennent à leur tour des "documents". Tout ceci est immédiatement compréhensible et la maîtrise de l'ordinateur en est simplifiée. Avec les métaphores, l'utilisateur reconnaît au lieu de devoir apprendre. Les utilisateurs se souviennent généralement plus facilement d'un objet que d'une commande.

La cohérence

C'est un des aspects les plus importants de la conception d'un programme Windows. C'est principalement par un souci de cohérence que le logo Windows 95 a été développé. Si toutes les applications sont cohérentes dans leur manière de présenter des données et dans leur façon d'interagir avec l'utilisateur, ce dernier passe son temps à accomplir les tâches, et pas à apprendre à communiquer avec votre application. Cette cohérence doit s'exprimer sur plusieurs points :

- Assurez-vous que votre application fonctionne comme le système d'exploitation Windows. Si c'est le cas, l'utilisateur peut facilement mettre à profit ses connaissances de Windows dans votre application.
- Vérifiez la cohérence interne de votre application. Si Ctrl + C sert de raccourci pour l'impression sur un écran, n'utilisez pas Ctrl + D sur un autre écran.
- Assurez-vous de la cohérence de vos métaphores. Si une icône Trou noir est la même que la Corbeille, les utilisateurs penseront qu'ils peuvent aussi récupérer les documents dans cette icône (alors que votre Trou noir fait disparaître définitivement vos fichiers).

Nous reviendrons sur la cohérence un peu plus loin dans cette leçon lorsque nous parlerons des menus, des barres d'outils et des autres contrôles usuels.

Etre magnanime

On passe souvent un bon moment à explorer une nouvelle application, à appuyer sur les boutons pour voir ce qui se passe. Dans une application bien écrite, cela ne pose jamais de problème. Si l'on est sur le point d'effectuer une action qui formatera le disque dur, une boîte de

dialogue apparaît et prévient de ce qui risque d'arriver. Il suffit alors d'appuyer sur le bouton Annuler qui est obligeamment proposé, et rien n'est perdu.

C'est là le concept de magnanimité. Vous devez laisser les utilisateurs explorer votre application. Toutes les actions de l'utilisateur doivent être réversibles ou corrigibles. Les utilisateurs doivent être prévenus *à l'avance* de l'aspect potentiellement dangereux d'une action qu'ils s'apprêtent à effectuer. Il arrive aussi que l'utilisateur clique au mauvais endroit. L'erreur est humaine. Vous devez en tenir compte et exiger confirmation de toute action potentiellement destructive, au cas où il s'agirait d'une erreur.

Note

Vous pouvez éventuellement donner à l'utilisateur la possibilité de désactiver certains messages de confirmation une fois qu'il est habitué au fonctionnement de l'application. Une boîte de dialogue d'options à onglets permettant à l'utilisateur de désactiver la confirmation des suppression de clients, de fichiers, etc. serait bien utile dans le cadre d'une utilisation professionnelle de votre application.

Du répondant

Il n'y a rien de plus irritant qu'un ordinateur qui fait son mystérieux. On ne sait pas ce qu'il est en train de faire, et il n'indique rien. Une règle d'or : faites toujours savoir à l'utilisateur ce qui se passe. Tenez-le informé régulièrement. Vous pouvez combiner des indices visuels et sonores pour montrer à l'utilisateur que vous ne l'avez pas oublié.

Il est important que la réaction s'exprime près de l'endroit où l'utilisateur travaille. S'il entre des données au sommet de l'écran, n'affichez pas un message d'erreur en bas (à moins qu'il n'y ait une barre d'état). Vous pouvez changer la forme du curseur pour indiquer une condition particulière (comme le tristement célèbre sablier qui indique que le système est occupé). Un utilisateur ne supporte pas que son ordinateur fasse le mort plus de quelques secondes.

De l'esthétique

Votre application doit aussi être visuellement agréable. Cela suppose plusieurs choses. En plus d'utiliser les couleurs systèmes pour votre écran (pour que votre application se fonde dans l'environnement), le design de l'écran lui-même est de toute importance. La disposition des objets détermine la facilité d'usage de votre écran, de même que le nombre d'objets sur votre écran. Dans la mesure du possible, respectez la "règle de sept". Ne donnez que sept choix à l'utilisateur (à deux près). Ce nombre de cinq à neuf choix découle de recherches sur le nombre de choses que le cerveau peut embrasser simultanément. Avec plus de neuf objets, l'utilisateur s'embrouille. Nous reviendrons sur la conception de l'écran dans la suite de cette partie.

De la simplicité

Le dernier principe de conception est la simplicité. Votre application doit être facile d'apprentissage et d'usage. Vous devez trouver un équilibre entre deux concepts opposés : permettre l'accès à toutes les fonctions et informations de l'application tout en gardant l'interface et la

manipulation de l'application aussi simple que possible. Une bonne application sait ménager ces deux principes.

Ne soyez pas trop verbeux quand vous créez vos écrans. Dans vos libellés de champs d'entrée de données, écrivez "Nom" et pas "Le nom de famille du client". Essayez de vous exprimer en un minimum de mots sans pour autant perdre en sens. Microsoft recommande également le concept de révélation progressive. En d'autres termes, présentez les données seulement lorsque c'est nécessaire. Ainsi, dans un programme de répertoire, vous pouvez montrer le nom et le numéro de téléphone d'une personne sur l'écran initial, et laisser l'utilisateur appuyer sur un bouton s'il désire la totalité des informations concernant la personne.

Conception centrée sur les données

Vous avez sans doute déjà utilisé sans le savoir une application dont la conception est centrée sur les données.

Nouveau

Une conception centrée sur les données signifie que l'utilisateur peut manipuler des données spécifiques sans devoir faire appel à des éditeurs ou des applications extérieures. Lorsque l'utilisateur commence à agir sur les données (pour les visualiser ou les éditer), les outils adéquats deviennent automatiquement disponibles.

Ce concept est appliqué dans des applications telles que Microsoft Word. Ainsi, lorsque vous cliquez sur un dessin, la barre d'outils Dessin apparaît au bas de l'écran, proposant tous les outils nécessaires à la manipulation du dessin.

Le concept de document permet de bien mettre en place les concepts dans l'esprit de l'utilisateur. La vision des choses centrée sur le document est facile à mémoriser. Et les documents ne se limitent pas aux applications de traitement de texte. De nombreuses applications autres que des traitements de texte utilisent des documents. Dans certains logiciels de communication, les transferts de fichiers sont même appelés "transferts de documents".

Quel modèle pour son application ?

Nous devons maintenant parler de la gestion des fenêtres. Il existe deux modèles d'application différents, l'interface de document unique (SDI en anglais) et l'interface de documents multiples (MDI en anglais). En choisissant Fichier/Nouveau/Projets, puis application SDI ou application MDI, vous pouvez créer un squelette d'application. Si vous n'avez jamais eu affaire à des applications MDI, vous ne savez pas forcément quels sont les avantages du MDI sur le SDI, ni quels sont les critères vous permettant de choisir une méthode plutôt qu'une autre.

Dans la quasi-totalité des cas, une application peut être interfacée avec l'utilisateur par le biais d'une fenêtre primaire unique. Si des informations supplémentaires doivent être affichées ou

collectées, une fenêtre secondaire peut être utilisée. Un bon exemple d'application SDI est l'Explorateur. L'Explorateur comporte une unique fenêtre primaire qui contient pratiquement toutes les informations nécessaires à son utilisation. Lorsque vous consultez des propriétés, une fenêtre secondaire apparaît. Avec une application SDI il est plus facile pour vous, le développeur, de gérer la relation exclusive entre l'écran et les données qu'il contient.

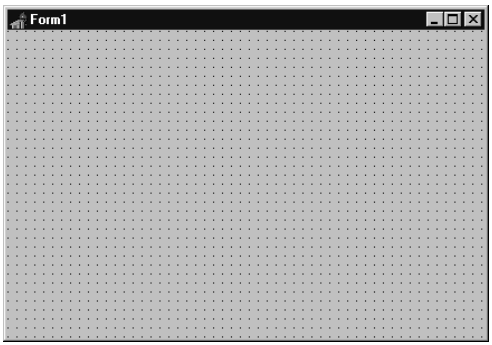
Une application MDI comporte aussi ses avantages. Microsoft Word est un bon exemple d'application MDI. Une application MDI a une fenêtre parent (la fenêtre primaire) et plusieurs fenêtres enfant (aussi appelées documents). Il arrive qu'il soit plus efficace d'afficher des informations dans plusieurs fenêtres qui partagent des éléments d'interface (comme une barre d'outils ou une barre de menus). Les fenêtres de document sont contrôlées ou découpées par le parent. Si vous réduisez la taille de la fenêtre parent, les fenêtres enfant peuvent être masquées.

Dans quel cas doit-on utiliser une application MDI ? Ces cas sont rares. Tout d'abord, MDI ne doit être utilisé que lorsque toutes les fenêtres enfant seront utilisées pour contenir des objets identiques (tels que des documents ou des feuilles de calcul). N'utilisez pas MDI si vous comptez avoir différents types de fenêtres enfant (des documents et des feuilles de calcul ensemble par exemple). N'utilisez pas MDI si vous souhaitez décider de la fenêtre qui sera au sommet en utilisant la propriété "rester au sommet", si vous souhaitez contrôler la taille des enfants, ou encore si vous souhaitez pouvoir masquer et montrer une fenêtre enfant. MDI a été conçu pour un type très particulier d'application, comme Word ou Excel, pour qui les enfants sont tous uniformes. Essayer de "forcer" une implémentation MDI sur une application d'un autre genre rendra malheureux développeur et utilisateurs. Enfin, il convient de remarquer que Microsoft n'encourage pas le développement de nouvelles applications MDI (en grande partie à cause du grand nombre de piètres applications MDI développées sous les versions précédentes de Windows).

Les composants d'une fenêtre

Nous allons revoir ensemble les bases de l'interface. Une fenêtre type est constituée d'un cadre et d'une barre de titre identifiant le nom de l'application dans la fenêtre, ou l'objet actuellement visualisé dans la fenêtre, comme indiqué Figure 7.1. Si l'objet visualisé est plus grand que la fenêtre, des barres de défilement apparaissent pour que l'utilisateur puisse faire défiler l'ensemble de la fenêtre.

Figure 7.1
Une fenêtre générique.



Le cadre de la fenêtre (s'il est dimensionnable) doit inclure une poignée de dimensionnement située dans le coin en bas à droite. En plus de la barre de titre, plusieurs autres éléments peuvent se trouver sur la fenêtre, tels que des menus, des barres d'outils, des barres d'état et d'autres encore. Examinons ensemble chacun de ces éléments, en insistant sur la façon dont on doit les construire pour qu'ils satisfassent les exigences d'une bonne conception.

Icônes de barre de titre

Lorsque vous construisez une application Windows, votre fenêtre primaire doit contenir une version réduite de l'icône de votre application. Cette icône représente le produit, si ce produit n'est pas un utilitaire ou un outil qui crée ou visualise un document d'aucune sorte. La Figure 7.2 montre une barre de titre standard.

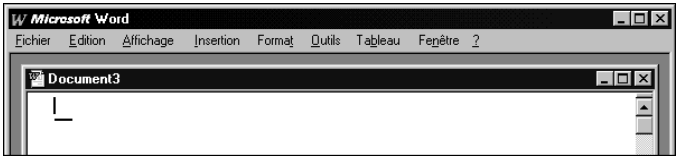
Figure 7.2
Barre de titre d'utilitaire.

Si la fenêtre primaire est utilisée pour visualiser des documents, il est préférable de placer dans le coin supérieur gauche de la fenêtre une icône de document plutôt que celle de l'application. Placez cette icône, même si l'utilisateur n'a pas enregistré le document qu'il crée ou visualise. Il s'agit là d'une question de cohérence. Nous devons faire en sorte que l'application soit uniforme et cohérente dans sa représentation des données et dans sa forme.

Figure 7.3
Barre de titre d'outils pour un document.

Les interfaces de documents MDI sont un cas particulier. Dans une application MDI, l'icône de l'application se trouve dans la fenêtre primaire et l'icône de document est placée dans toutes les fenêtres enfant. Delphi 3 se charge lui-même d'un grand nombre d'éléments par défaut concernant les fenêtres et vous décharge généralement de ces considérations.

Figure 7.4
Une application MDI.



Texte de la barre de titre

La fonction principale d'une barre de titre est d'identifier le nom du document visualisé dans la fenêtre. Ainsi, par exemple, si l'utilisateur ouvre un document appelé "Mon CV" dans une application MDI, la barre de titre devra comporter l'icône de document représentant le type de document concerné, suivie du nom du document, "Mon CV" en l'occurrence. De plus, vous pouvez inclure le nom de l'application à la suite du nom du document. Si vous faites figurer le nom de l'application, séparez par un tiret le nom du document et celui de l'application. La Figure 7.5 en montre un exemple classique.

Figure 7.5
Exemple de l'ordre des textes dans une barre de titre.

Si l'application est un utilitaire ou un outil et n'a pas de nom de document qui lui soit associé, vous pouvez utilisez l'icône de l'application, suivie par le seul nom de l'application. La calculatrice est un bon exemple d'une telle application. Dans le cas d'une application comme l'Explorateur (qui permet de visualiser de nombreux types d'objets), vous pouvez placer le nom de l'application, suivi d'un texte spécifiant le type de données en cours de visualisation.

Là encore les applications MDI sont un cas à part. Dans une application MDI, vous affichez le nom de l'application dans la fenêtre parent, et le nom du document dans la fenêtre enfant. Lorsque l'utilisateur agrandit l'enfant, la fenêtre parent suit les mêmes conventions que les utilitaires, en faisant figurer le nom de l'application, puis un tiret et le nom du document.

Nouveaux documents

Plusieurs questions sont à prendre à considération au sujet du titrage des fenêtres. Si votre application permet à l'utilisateur de créer de nouveaux documents, votre application doit automatiquement nommer les fenêtres à l'aide du type de fichier suivi d'un identificateur unique. Par exemple, l'usage de document(1) ou feuille(1) (comme c'est le cas dans Windows et Excel)

est un moyen rapide de poursuivre les opérations immédiatement après la création d'un document par l'utilisateur. Vous ne devez pas demander le nom d'un document lors de la création. C'est au moment où l'utilisateur enregistre son document pour la première fois qu'il faut lui demander un nom.

Par ailleurs, lorsque votre application crée la fenêtre (et le document) temporaire, il est important que son nom soit unique. C'est la raison pour laquelle la plupart des applications se contentent d'incrémenter le chiffre qui se trouve à la fin du nom de fichier (document1, document2, etc. par exemple). Il est primordial que le nom ne soit pas le même que celui d'une fenêtre déjà existante. Le nom temporaire (document1) doit être le titre de la fenêtre de document jusqu'à ce que l'utilisateur lui fournisse un nom définitif. Le nom temporaire sera alors remplacé par celui fourni par l'utilisateur.

Boutons de la barre de titre

Les derniers éléments de la barre de titre que nous allons évoquer sont les boutons. Windows 95 comporte un nouvel ensemble de boutons qui peuvent prêter à confusion pour les utilisateurs de Windows 3.1. Les boutons qui sont maintenant pris en charge par Windows 95 sont décrits dans le tableau qui suit :

Bouton	*Commande*	*Fonction*
	Fermer	Ferme la fenêtre courante
	Réduction	Réduit la fenêtre courante
	Agrandissement	Agrandit la fenêtre courante
	Restauration	Restaure la fenêtre courante à sa taille initiale, suite à un agrandissement ou à une réduction

Le bouton qui pose le plus de problèmes aux utilisateurs de Windows 3.1 est le bouton Fermer. Le symbole X semble signifier Réduction pour certains. Pourtant, ce bouton ferme la fenêtre courante, comme le fait Alt + F4. Le bouton Fermer doit toujours être le bouton le plus à droite, et doit être séparé des autres par un espace plus grand. Le bouton Réduction doit toujours précéder le bouton Agrandissement, et le bouton Restauration doit remplacer le bouton Réduction ou Agrandissement (selon le bouton qui vient d'être utilisé). En tant que programmeur, vous pouvez décider si ces boutons apparaissent sur vos fiches en modifiant les attributs `BorderIcons` pour cette fiche.

Ouvrir et fermer des fenêtres

Une des fonctionnalités les plus agréables dans une application Windows est la possibilité d'enregistrer vos paramètres lorsque vous quittez l'application. Vous pouvez ainsi enregistrer la taille et la position de la fenêtre d'application. Au prochain lancement du programme, la fenêtre retrouvera la même taille et la même position. Vous pouvez faire vos propres entrées dans le registre pour stocker les informations concernant taille et position. Une autre méthode consiste à prendre la taille de l'écran et à faire en sorte que la fenêtre de votre application occupe la taille de l'écran, ou qu'elle n'en occupe que la moitié et soit centrée sur l'écran.

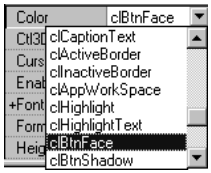
Le comportement d'une application Windows dépend dans une large mesure de sa conception. Ainsi, un programme de type Bloc-notes permettra que plusieurs instances du même programme s'exécutent simultanément. En revanche, dans le cas d'une application telle que Delphi 3, si l'utilisateur lance une autre copie, l'instance en cours d'exécution est tout simplement mise en avant-plan. Ce type de réponse est bien adapté aux applications pour lesquelles une seule instance peut s'exécuter à la fois. Vous pouvez également faire apparaître une boîte de dialogue donnant la possibilité à l'utilisateur de faire venir au premier plan l'application en cours d'exécution ou de lancer une autre application.

Couleurs de fenêtre

La couleur peut apporter un plus à votre application. Mais n'exagérez pas. Delphi 3 fournit les couleurs système dans la palette, vous pouvez ainsi faire en sorte que les couleurs de votre application s'accordent avec la palette employée dans le reste du système. Si vous cherchez les couleurs dans l'Inspecteur d'objets, vous en trouverez certaines comme `clWindowActive`. Ce sont les couleurs système courantes qui ont été choisies par l'utilisateur. Si vous décidez que ces couleurs seront celles de votre application, toute modification des couleurs système par l'utilisateur entraînera les mêmes modifications pour les couleurs de votre application. Vous pouvez aussi employer des couleurs *statiques* (les couleurs de votre application ne changeront pas même si l'utilisateur modifie les couleurs système). Lorsque vous créez une nouvelle fiche ou ajoutez des composants à une fiche, Delphi 3 attribue des couleurs à ces éléments conformément à la palette de couleurs Windows 95. Vous devez changer les couleurs manuellement à l'aide de l'Inspecteur d'objets si vous souhaitez imposer vos propres décisions chromatiques.

Il n'est cependant pas recommandé de substituer vos propres couleurs aux couleurs système, et ce pour plusieurs raisons. Les choix de couleurs de l'utilisateur sont peut-être motivés (pour économiser l'énergie par exemple, puisque des fonds noirs tirent moins sur la batterie, ou pour des raisons de visibilité, l'utilisateur choisissant des nuances de gris car il est daltonien). Dans le doute, mieux vaut s'abstenir. Si vous modifiez les couleurs dans votre application, essayez de vous en tenir aux 16 couleurs de base. Si vous essayez d'utiliser 256 couleurs (ou pire, 16 millions...), cela risque de ralentir votre application et de donner des résultats hasardeux chez les utilisateurs ne disposant que de 16 couleurs.

Figure 7.6
La palette de couleurs de l'Inspecteur d'objets.



Menus

Nous avons tous déjà utilisé des menus dans une application Windows. Ils permettent un accès facile aux fonctionnalités d'un programme, accès basé sur la reconnaissance (nous sommes tous habitués au célèbre Fichier/Ouvrir) plutôt que sur l'apprentissage d'obscurs raccourcis clavier. Il existe différents types de menus, parmi lesquels les menus contextuels, les menus déroulants et les menus en cascade. Examinons chacun de ces types de menus.

Le menu déroulant existe principalement sur les barres de menus. La barre de menus se trouve dans la plupart des applications. Elle contient des éléments appelés titres de menus. Ces titres, lorsqu'ils sont sélectionnés, permettent d'accéder à des menus déroulants. Ces menus déroulants contiennent le niveau de sélection suivant, c'est-à-dire les éléments de menu. La Figure 7.7 montre une barre de menus standard.

Figure 7.7
Une barre de menus classique.

Les contrôles visuels fournis avec Delphi 3 vous permettent de construire votre propre barre de menus ainsi que les menus déroulants qui lui sont associés. Les choix de menus que vous donnez à votre utilisateur dépendent des fonctionnalités de votre application. Vous pouvez même donner la possibilité à l'utilisateur de modifier ou de personnaliser les menus de votre application. Si vous le faites, assurez-vous que vous donnez accès à un ensemble de choix standard par le biais d'un mécanisme standard, sous forme de menus contextuels et de barres d'outils par exemple. De cette manière, l'utilisateur ne courra pas le risque de personnaliser l'interface au point de ne plus pouvoir s'en servir...

Astuce

Si vous sélectionnez Fichier/Nouveau/Onglet Projet/application SDI, vous obtenez une structure de menu minimale, comportant les commandes Fichier et Aide. Si vous créez une application MDI, vous obtenez une barre de menus plus étendue, comportant les titres de menu Edition et Fenêtre.

Le menu Fichier

Le menu Fichier est le moyen principal d'accès de l'utilisateur aux commandes les plus importantes de votre application. En général, vous y ferez figurer les commandes Nouveau, Ouvrir, Enregistrer et Enregistrer sous, si votre application ouvre des documents de tout type. Une nouvelle commande de Windows 95 et NT 4.0 est la commande Envoyer vers qui peut aussi figurer dans notre menu. Si votre application permet l'impression, une commande Imprimer a toute sa place dans ce menu. Enfin, si votre application prend en charge la commande quitter, ce doit être la dernière sélection du menu. Si l'application reste active même lorsque la fenêtre est fermée (comme dans le cas du Contrôle de volume de la barre de tâches), utilisez la commande Fermer plutôt que Quitter.

Il est important que les éléments de menu se trouvent toujours à la même place pour que l'utilisateur se trouve en terrain connu même face à une nouvelle application. Fichier/Quitter ou Fichier/Imprimer sont des réflexes habituels chez l'utilisateur, et il est bon qu'ils le restent.

Note

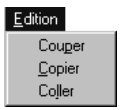
Vous avez peut-être remarqué que chaque commande de menu possède une lettre soulignée ou une combinaison de touches associées, comme illustré Figure 7.8. Vous apprendrez à définir ses accélérateurs un peu plus loin dans cette leçon.

Figure 7.8
Un menu Fichier classique.

Le menu Editer

Le menu Editer est indispensable dans des applications permettant d'éditer ou de visualiser des documents. Ce menu contient généralement les commandes Couper, Copier et Coller, comme illustré Figure 7.9. Vous pouvez placer dans ce menu des commandes portant sur les objets OLE ainsi que d'autres commandes plus sophistiquées. Ces commandes peuvent comprendre Annuler (pour revenir sur la dernière action effectuée), Répéter (pour répéter la dernière action), Chercher et Remplacer (pour chercher un texte ou le remplacer par un autre), Supprimer (pour supprimer l'élément sélectionné) et Dupliquer pour créer une copie de l'élément sélectionné. La commande Sélectionner tout est elle aussi bien utile pour sélectionner l'ensemble d'un document.

Figure 7.9
Un menu Edition standard.



Le menu Affichage

Le menu Affichage permet à l'utilisateur de modifier la façon de visualiser les données. Cette modification peut consister en un zoom avant ou arrière, ou en la visualisation d'éléments supplémentaires (tels qu'une règle ou une barre d'outils). Dans ce menu, vous choisissez les éléments, une Règle par exemple, en les cochant pour indiquer qu'ils ont été sélectionnés.

Figure 7.10
Un menu Affichage classique.

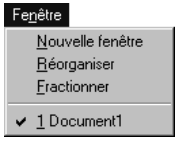
Le menu Fenêtre

Le menu Fenêtre n'est généralement présent que dans les applications MDI, où il s'avère nécessaire pour gérer plusieurs fenêtres. Ce menu comporte généralement les commandes Nouvelle fenêtre, Mosaïque, Cascade, Fractionner et Réduire tout. Dans ce menu, vous pouvez également permettre à l'utilisateur d'accéder à toutes les fenêtres ouvertes. Cela s'effectue généralement au moyen d'une liste des fenêtres placée au bas du menu déroulant. Cette liste permet à l'utilisateur d'accéder à n'importe quelle fenêtre en choisissant son nom dans le menu.

Astuce

Delphi 3 inclut automatiquement la liste des fenêtres ouvertes dans le squelette MDI standard généré avec Fichier/Nouveau/Onglet Projets/Application MDI.

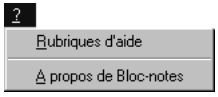
Figure 7.11
Un menu Fenêtre classique.



Le menu Aide

Un des menus les plus importants est le menu Aide. Ce menu vous permet de faire figurer en ligne les réponses à la plupart des questions que se pose l'utilisateur. Comme partie standard de l'aide Windows, vous devez fournir une commande Rubriques de l'aide qui permet d'accéder au navigateur des Rubriques d'aide, ainsi qu'à d'autres éléments tels que la Recherche de rubriques et d'éventuels Assistants d'aide (comme dans Word). Si vous souhaitez inclure des informations sur le numéro de version ou sur votre société, faites-les figurer dans une boîte de dialogue A propos, accessible par la sélection de menu Aide/A propos. N'oubliez pas : plus votre aide sera performante, moins vous aurez d'appels de clients. La Figure 7.12 donne un exemple de menu d'aide.

Figure 7.12
Un menu Aide classique.



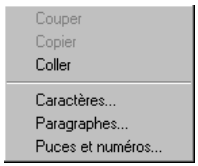
Menus contextuels

Les menus contextuels n'étaient pas véritablement passés dans les mœurs avant l'arrivée de Windows 95. Le bouton droit de la souris a maintenant droit de cité. En cliquant à droite sur un objet, l'utilisateur se voit proposer un ensemble de commandes destinées à la manipulation de cet objet. Par exemple, lorsque vous cliquez à droite sur le fond de Windows 95, un menu contextuel apparaît vous proposant toutes les fonctions relatives à l'affichage.

Le côté agréable des menus contextuels est qu'ils ne proposent que les commandes qui sont adaptées à l'objet sélectionné. Vous devez toujours inclure un menu contextuel si vous avez une barre de menus (parce que certains des éléments de la barre de menus risquent forts de s'avérer adaptés à certains objets de votre application).

En ce qui concerne l'organisation des menus contextuels, tâchez de faire figurer un minimum d'éléments dans le menu lui-même. Les sélections doivent être courtes et les propriétés individuelles ne doivent jamais figurer dans le menu lui-même. Faites toujours figurer un choix de menu Propriétés et laissez l'utilisateur passer dans un autre écran pour les consulter. On tolère l'existence dans le menu contextuel de commandes qui ne sont pas présentes dans la barre de menus (et vice versa), mais assurez-vous que le menu contextuel n'est pas le seul moyen d'accéder aux commandes de votre application. La Figure 7.12 donne un exemple de menu contextuel.

Figure 7.13
Un menu contextuel classique.



L'ordre des éléments est primordial (toujours dans ce même souci de cohérence). En partant du sommet, les premières commandes doivent être les fonctions primaires qui sont habituellement effectuées sur cet objet, telles qu'Ouvrir, Explorer, Exécuter, Imprimer et Lecture. Les commandes de transfert, telles que Envoyer vers, viennent ensuite, de même que l'éventuelle commande "Qu'est-ce que c'est ?". Couper, Copier et Coller doivent être placées dans cet ordre. Enfin, la commande Propriétés doit toujours être le dernier élément de votre menu contextuel.

Les menus contextuels sont très souvent utilisés dans l'environnement Windows. Si vous cliquez à droite sur le fichier exécutable de votre application dans l'Explorateur, il vous sera proposé plusieurs choix. Il vous est ainsi possible d'ajouter des éléments de menu spécifiques destinés à prendre en charge votre application. Ainsi, si vous installez WinZip pour Windows 95 (un produit indispensable), vous pouvez constater que WinZip place une commande Add to Zip (ajouter à un fichier zippé) dans l'Explorateur, vous évitant ainsi de devoir lancer WinZip pour décompresser un fichier. C'est là un exemple de la façon dont les menus contextuels rendent les utilisateurs plus efficaces.

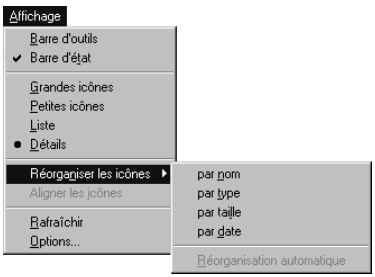
Sous-menus

Vous avez sans doute déjà utilisé une application comportant des sous-menus en cascade (Delphi 3 par exemple). Les menus enfant (un autre nom désignant les sous-menus en) permettent d'éviter de surcharger un menu d'informations. Regardez par exemple tous les choix qui se trouvent dans les divers sous-menus Fichier de Delphi 3. Imaginez un peu la confusion du menu si tous les éléments étaient placés au même niveau. La Figure 7.14 donne un exemple de sous-menu.

Pour voir un bon exemple d'utilisation de menus enfants, ouvrez Delphi 3. La présence d'un sous-menu est indiquée par une flèche triangulaire qui suit un élément de menu dans le menu parent (regardez le choix de menu Fichier/Réouvrir par exemple).

Attention *N'abusez pas des sous-menus. Ils tendent à rendre les choses plus complexes en obligeant l'utilisateur à manipuler plusieurs fois la souris. Même Delphi 3 en fait un usage modéré.*

Figure 7.14
Un sous-menu classique.



Essayez également de limiter le nombre de niveaux. N'enfouissez pas de commandes importantes sous cinq niveaux de menus. L'utilisateur risquerait de perdre un temps précieux.

Libellés de menu, raccourcis, touches d'accès et types

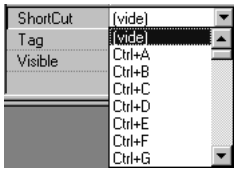
Lorsque vous créez vos menus, vous devez également vous intéresser à quelques concepts supplémentaires. Aux libellés des menus, tout d'abord. Les éléments de la barre de menus principale et des sous-menus doivent être dans la mesure du possible des identificateurs tenant en un mot. Les noms doivent être succincts et parlants. N'utilisez pas de phrases de plus de deux mots car le menu gagnerait en complexité et l'utilisateur devrait passer plus de temps à lire le menu pour effectuer ses choix.

En plus de la souris, il existe deux autres méthodes d'accès aux sélections de menu : les raccourcis et les combinaisons de touche.

Nouveau *Un raccourci clavier est une combinaison de touches (Ctrl + F4 par exemple), que vous affectez pour exécuter une commande de menu.*

Les raccourcis permettent par exemple à l'utilisateur d'effectuer un "enregistrement rapide" du document courant en appuyant sur F2. Vous pouvez affecter des raccourcis clavier à n'importe quelle tâche, du moment que vous ne modifiez pas l'affectation de combinaisons usuelles telles que Alt + F4 (fermeture de la fenêtre courante). Dans le contrôle visuel menu de Delphi 3, vous pouvez affecter des raccourcis clavier en modifiant l'attribut shortcut de chaque élément de menu. Delphi vous propose une liste de tous les raccourcis clavier disponibles et il vous suffit d'en choisir un. La Figure 7.15 donne un exemple de menu d'aide.

Figure 7.15
La boîte Propriétés ShortCut.



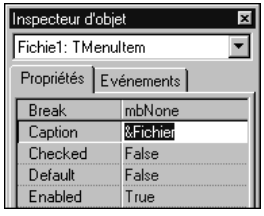
Parallèlement aux raccourcis clavier, vous pouvez utiliser des touches de raccourcis.

Nouveau *Les touches de raccourcis (à ne pas confondre avec les raccourcis clavier) sont utilisées en combinaison avec la touche ALT du clavier. Ainsi, dans Delphi, vous pouvez accéder à la boîte de dialogue Nouveaux éléments en appuyant sur Alt + F, N.*

Note *Lorsque vous utilisez des touches de raccourcis dans différents niveaux de menu, la touche Alt ne doit être enfoncée que pour la première touche d'accès. Cette première sélection effectuée, il suffit d'appuyer sur les lettres elles-mêmes.*

Les touches que vous pouvez utiliser en combinaison avec la touche ALT sont soulignées. Si vous voyez le mot Fichier comportant un F souligné, vous pouvez utiliser la combinaison Alt + F. Lorsque vous utilisez le contrôle visuel menu, vous pouvez placer un caractère d'accès (le caractère souligné) dans le titre en plaçant une perluette (&) devant la lettre de l'élément de menu que vous souhaitez souligner. Ainsi, le mot Fichier dont le F serait souligné s'écrirait &Fichier. La Figure 7.16 donne un exemple.

Figure 7.16
La propriété Caption *de l'élément de menu.*



De nombreux facteurs doivent être pris en considération lorsque vous créez le système de menus de votre application. Nous avons abordé la plupart de ces facteurs. Des tests d'ergonomie permettent par ailleurs de mettre à jour les faiblesses de conception. N'hésitez pas à solliciter l'avis de ceux à qui est destiné le produit.

Contrôles

Pour que votre application soit réellement utile, elle doit interagir avec l'utilisateur. Les contrôles s'en chargent. Les contrôles permettent à l'utilisateur de contrôler les actions ou les propriétés des objets de l'application. Les contrôles revêtent des formes et des styles variés.

On active généralement les contrôles en utilisant le pointeur de la souris en conjonction avec le bouton gauche de la souris. La plupart des contrôles comportent ce que l'on appelle une "zone de réaction". Cette zone de réaction est la zone qui est sensible aux clics gauches de la souris. Si le pointeur arrive dans cette zone et qu'un clic gauche survient, le contrôle est activé. Certaines zones de réaction sont très visibles sur les contrôles, comme par exemple la zone de réaction d'un bouton qui correspond à la zone délimitée par les bordures du bouton. D'autres contrôles peuvent comporter des zones de réaction qui sont plus grandes ou plus petites que le contrôle lui-même. Une case à cocher, par exemple, comporte une zone de réaction plus grande que la case à cocher, car elle recouvre aussi le libellé de la case.

Les libellés sont importants car ils permettent d'indiquer immédiatement à l'utilisateur quelle est la fonction du contrôle. Les libellés, tout comme les éléments de menu, doivent être concis et parlants. Comme dans les menus, il est important de fournir des touches d'accès aux contrôles pour que l'utilisateur puisse utiliser directement le clavier, sans passer par la souris.

Boutons de commande

Plusieurs types de boutons sont utilisés dans Windows 95. Le premier type de bouton est le bouton de commande. Il permet d'activer une commande associée à ce bouton. En général, il faut appuyer sur le bouton. Lorsque le bouton est relâché, si le pointeur de la souris se trouve sur le bouton, la commande associée au bouton est exécutée. Si le pointeur n'y est pas, la commande n'est pas exécutée. Ceci donne une deuxième chance à l'utilisateur, lui permettant d'annuler la commande même s'il a déjà appuyé sur le bouton.

Cette deuxième chance n'est donnée que si vous utilisez la souris pour activer le bouton. Si le bouton a le focus et que vous appuyez sur Entrée, il n'y a plus d'échappatoire : il a été activé.

Note

Lorsque nous utilisons le terme appuyer à propos des boutons d'une fiche, on parle en fait de deux choses différentes. A l'aide de la souris, on peut appuyer visuellement sur un bouton ou l'activer. Le bouton apparaît enfoncé lorsque vous cliquez sur le bouton de votre souris. Vous pouvez également appuyer sur le bouton (l'activer) lorsque le bouton a le focus, en appuyant sur la touche Entrée. Ces deux actions ont le même résultat : elles permettent d'appuyer sur le bouton.

En plus du bouton de commande, Windows 95 fait grand usage du bouton Déplier. Il s'agit d'un bouton de commande qui élargit une fenêtre lorsqu'il est activé. Lorsque vous utilisez un bou-

ton de commande comme le bouton Déplier, placez le signe >> pour indiquer que ce bouton élargira la vue actuelle. Ceci permet à l'utilisateur de voir une partie des informations, puis d'appuyer sur le bouton Déplier s'il désire en voir la totalité.

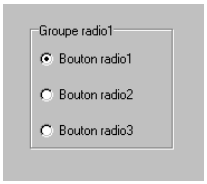
Si votre bouton de commande demande des informations supplémentaires pour que la commande qui lui est associée puisse s'exécuter correctement, placez des points de suspension (...) après le libellé du bouton de commande. Ceci indique à l'utilisateur qu'il devra fournir des informations supplémentaires (dans une boîte de dialogue la plupart du temps). Lorsque des boutons de commande ne sont pas disponibles, ils doivent être grisés. Ceci est effectué en définissant comme `False` la propriété `Enabled` de ce bouton.

Boutons radio

Les boutons radio aident l'utilisateur à sélectionner des options dans votre application. C'est pourquoi on les appelle parfois des boutons d'options. Les boutons radio apparaissent sous forme de petits cercles et doivent être disposés par groupes de deux à sept. Si vous avez besoin de plus d'éléments, il est préférable d'utiliser d'autres contrôles, tels que des boîtes de liste.

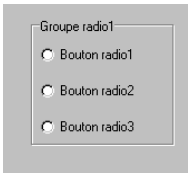
Les boutons radio peuvent être présentés de deux manières. La première est un mode exclusif (le mode le plus usuel, représenté Figure 7.17). En mode exclusif, un seul bouton du groupe peut être sélectionné à la fois. Ce mode est parfait pour des options mutuellement exclusives.

Figure 7.17
Des boutons radio en mode exclusif.



L'autre état possible est le mode dit de valeurs mélangées. Dans ce mode, on peut sélectionner simultanément plusieurs boutons radio. Ce mode est idéal pour la sélection d'attributs de fichier, puisqu'un fichier peut être en lecture seule, caché et de type archive tout à la fois.

Figure 7.18
Des boutons radio en mode mixte.



Beaucoup d'utilisateurs pensent que les boutons radio sont en mode exclusif, ce qui signifie qu'un bouton est toujours coché. Lors de la conception de l'application, le programmeur choisit généralement le bouton coché. Ceci s'effectue en définissant comme `True` la valeur de l'attribut d'un des boutons. Pour créer un groupe de boutons en mode exclusif, il suffit de les placer dans un contrôle `RadioGroup` (Groupe radio), `Panel` (Volet) ou `Bevel` (Biseau). Ces contrôles vous

assurent qu'un seul bouton radio du groupe pourra être actif à un moment donné. Si vous souhaitez utiliser les boutons radio en mode valeurs mélangées, il suffit de ne pas les placer dans un contrôle groupe radio. Les boutons radio agiront ainsi indépendamment les uns des autres. Dans ce cas, vous pouvez définir comme `False` l'attribut de valeur de tous les boutons radio, en laissant à l'utilisateur la possibilité de les cocher.

Info *Il n'est pas recommandé d'utiliser des boutons radio en mode valeurs mélangées. Dans ce cas, il est préférable d'utiliser des cases à cocher.*

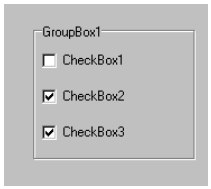
Comme pour les autres contrôles, il est important d'affecter des touches d'accès aux contrôles. Ainsi, l'utilisateur peut sélectionner un bouton radio spécifique directement à partir du clavier. Une autre méthode consiste à appuyer plusieurs fois sur la touche Tab jusqu'à ce que le contrôle soit mis en surbrillance, avant d'appuyer sur la barre d'espace pour le sélectionner.

Note *Il existe une autre méthode d'accès aux boutons radio. Si vous affectez une touche d'accès à la GroupBox, ou à un autre contrôle dont font partie les boutons radio, en appuyant sur la touche d'accès de ce contrôle de groupe, vous faites passer le focus sur les boutons radio de ce contrôle de groupe. Vous pouvez alors utiliser les touches de déplacement du clavier pour activer le bouton radio désiré.*

Cases à cocher

Grâce aux cases à cocher l'utilisateur peut sélectionner des options dans votre programme. La case à cocher peut prendre plusieurs états. Vous pouvez utiliser l'attribut `state` pour déterminer si la case est cochée ou non en définissant la propriété `State` comme étant `cbChecked` (état coché), `cbUnChecked` (état décoché) ou `cbGrayed` (état où la case est cochée mais grisée). La signification de ce dernier état est déterminée par le développeur lui-même. La Figure 7.19 vous donne un exemple de case à cocher.

Figure 7.19
Des cases à cocher dans tous les états possibles.



Lorsque vous utilisez des cases à cocher, regroupez-les par types de fonction. Ceci permet à l'utilisateur de deviner les relations qui existent entre les cases à cocher. Comme pour les autres contrôles, utilisez des touchez d'accès pour que l'utilisateur puisse interagir avec les cases à cocher par l'intermédiaire du clavier seul. Là encore, si vous cochez une case avec la souris, votre sélection peut être annulée en retirant la souris du contrôle avant de relâcher le bouton. Vous pouvez également griser la case à cocher en définissant comme `False` l'attribut `Enabled`. L'utilisateur ne pourra ainsi pas disposer du contrôle, quel que soit l'état de la case à cocher.

Boîtes de liste

Les boîtes de liste offrent un moyen facile et direct de présenter à l'utilisateur un grand nombre d'éléments. L'avantage avec une boîte de liste est que les éléments qui y figurent peuvent être de formes diverses et en nombre quelconque. A la différence des cases à cocher et des autres contrôles, si une sélection n'est pas disponible, elle n'apparaîtra pas du tout dans la boîte de liste. Il n'est pas question ici de contrôles grisés. Le contenu de la boîte de liste peut être disposé de diverses manières. Employez la méthode qui satisfait au mieux les besoins de votre utilisateur. Ainsi, disposez les noms dans l'ordre alphabétique, les nombres dans un ordre croissant et les dates dans un ordre chronologique.

Figure 7.20
Un exemple de boîte de liste.

Note

Il est important que vous utilisiez à bon escient les contrôles Boîtes de liste et Boîtes à options. Si vous avez une base de données qui fait ses 4 Go, ne chargez pas tout son contenu dans une malheureuse petite boîte de liste. Cette méthode est lente et mal adaptée. Imaginez un peu le temps que prendrait le défilement des 4 Go de données... Il est donc préférable dans ce cas d'utiliser un contrôle Edit et un bouton de recherche permettant d'affiner les choix disponibles.

Il n'y a pas de libellé associé à une boîte de liste. Si vous créez un libellé pour une boîte de liste, assurez vous qu'il est désactivé lorsque la boîte de liste est désactivée (en définissant comme `False` l'attribut `Enabled`). La version Delphi 3 de la boîte de liste comprend un attribut Multiselect qui permet d'utiliser cette boîte de liste comme une boîte à sélection unique ou à sélection multiple. Dans le cas d'une sélection unique, l'utilisateur ne peut sélectionner qu'un seul élément de la boîte de liste. Si l'attribut Multiselect est défini comme `True`, l'utilisateur peut sélectionner plusieurs éléments. Si la liste des éléments de la boîte de liste est plus longue que la hauteur de la fenêtre, une barre de défilement verticale apparaît.

Contrôles d'affichage

Il existe quatre types différents de contrôles d'affichage dans Windows 95. Le premier est le contrôle Affichage de liste. Ce contrôle permet d'afficher une liste d'éléments d'une manière similaire à la fenêtre droite de l'Explorateur. La liste peut être visualisée d'une de ces quatre façons :

- Icônes. Les éléments apparaissent sous forme d'icônes de grande taille sous lesquelles se trouvent les libellés.
- Petites icônes. Chaque élément apparaît sous forme d'une petite icône, à la droite de laquelle se trouve le libellé.
- Liste. Identique à Petites icônes, à ceci près que les icônes apparaissent en colonnes et sont ordonnées.

- Rapport. Les éléments sont disposés en colonnes. Toutes les colonnes sauf celle de gauche doivent être fournies par l'application qui affiche la boîte de liste. La liste des éléments n'est pas triée.

Les contrôles Affichage d'arborescence

L'affichage d'arborescence est une autre forme du contrôle affichage de liste. La différence principale est que l'affichage d'arborescence est plus adaptée à la présentation d'informations qui supposent une notion de hiérarchie. L'affichage d'arborescence vous permet d'associer des icônes à chaque élément de l'arborescence. L'icône d'un élément peut même changer lorsqu'il est réduit, pour faire contraste avec le même élément développé.

L'affichage d'arborescence permet de tracer des lignes entre des éléments et de bien suggérer les relations hiérarchiques qui existent entre les différents éléments. Un exemple typique d'utilisation de tels contrôles est l'Explorateur Windows. La fenêtre de gauche de l'Explorateur est un contrôle d'arborescence. Les icônes de lecteur représentent chaque lecteur, les icônes de dossiers représentent les dossiers appartenant aux différents lecteurs, et ainsi de suite. Ce contrôle permet de présenter clairement quelque chose qui aurait autrement pu laisser l'utilisateur perplexe.

Entrer et afficher du texte

On peut afficher du texte de deux manières avec Delphi 3. La première consiste à utiliser un contrôle de libellé pour afficher du texte statique. Cette méthode est fréquemment utilisée pour afficher des informations que l'utilisateur ne peut pas modifier, telles qu'un numéro de série ou de sécurité sociale.

Les spécifications Microsoft indiquent que nous devons nous assurer que le contrôle de libellé ne reçoit pas le focus lorsque l'utilisateur passe d'un contrôle à un autre à l'aide de la touche Tab. Delphi 3 pousse la sophistication un cran plus loin. Le contrôle de libellé ne possède pas d'ordre de tabulation, il ne peut donc pas recevoir le focus. De plus, si vous donnez l'attribut `FocusControl` au contrôle qui doit avoir le focus, et si l'utilisateur appuie sur la touche d'accès de ce contrôle, l'élément pointé par `FocusControl` reçoit le focus à sa place. Vous pouvez donc utiliser des contrôles de libellés et distribuer correctement le focus à ces contrôles.

La boîte d'édition est la deuxième manière d'afficher des données. L'avantage de ce contrôle est qu'il permet également d'éditer les données. Le contrôle d'édition de Delphi 3 comporte l'attribut `MaxLength` (Longueur max) qui vous permet de limiter la longueur de ce qu'entre l'utilisateur. Les techniques d'édition classiques, insertion, écrasement et suppression sont également prises en charge.

Le contrôle Memo est un contrôle dont la fonction est similaire. Il s'agit en fait d'un contrôle d'édition multiligne. Il comporte lui aussi une propriété `MaxLength`. A cette propriété s'ajoute la propriété `Lines` (Lignes) qui vous permet de définir la valeur de chaque ligne dans le contrôle.

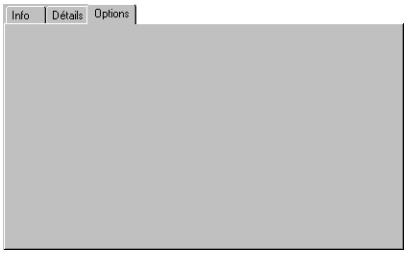
Contrôles de pages à onglets

La boîte de dialogue à onglets est la toute dernière nouveauté de Windows 95. Les onglets vous permettent d'organiser vos informations en différentes rubriques. Suivez pour les onglets les principes de nommage que nous avons évoqués au sujet des éléments de menu. Les pages à onglets elles-mêmes sont généralement alignées sur une seule rangée. Cependant, vous pouvez si nécessaire placer plusieurs lignes de pages à onglets sur une seule page. Chaque page à onglets contient des données auxquelles on accède en cliquant simplement sur l'onglet.

Note

Lorsque vous concevez une boîte de dialogue à onglets, vous devez prendre plusieurs choses en considération. Selon que vous placez vos boutons OK et Annuler sur la page à onglets elle-même ou sur la fiche qui contient les pages, vous indiquez des choses différentes à l'utilisateur. Si les boutons sont sur la page même, cela signifie que OK ne valide que les modifications effectuées dans cette page. Si le bouton OK se trouve sur la fiche, l'utilisateur doit pouvoir aller et venir entre les différentes pages pour y effectuer les modifications qu'il souhaite, avant d'appuyer sur le bouton OK pour valider l'ensemble des modifications.

Figure 7.21
Un exemple de page à onglets.



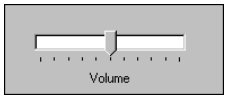
Curseurs

Les curseurs permettent de contrôler le réglage des valeurs de certaines données. Vous devez affecter une valeur minimum et maximum pour le curseur en utilisant les propriétés `Min` et `Max`. Ce contrôle est idéal pour ajuster des valeurs telles que le contraste, le volume, etc. Tout ce qui peut être réglé par un potentiomètre dans la vie réelle est un candidat idéal pour un curseur. Vous pouvez contrôler l'orientation du curseur (horizontale ou verticale), la présence d'incréments et leur nombre, et la taille du curseur lui-même.

Note

Même si le curseur a besoin de valeurs `Min` et `Max` définies, vous pouvez traduire ces valeurs pour refléter d'autres options. Ainsi, Windows 95 utilise le curseur dans la boîte de dialogue propriétés affichage pour définir la taille de l'écran. les incréments du curseur sont 640 × 480, 800 × 600 et 1024 × 768. Vous affectez une de ces trois valeurs du curseur pour qu'elles correspondent à des définitions de résolution d'écran. Grâce à cette conversion de valeur, le curseur trouve là une utilité de plus.

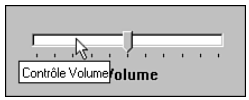
Figure 7.22
Un exemple de curseur.



Bulles d'aide

Les bulles d'aide (ou info-bulles) font partie des grandes innovations de Windows 95. Grâce à elles, l'utilisateur peut rapidement identifier des contrôles et leurs fonctions. Delphi 3 propose un "conseil" ou `Hint` (l'équivalent Delphi d'une info-bulle) pour tous les contrôles importants. Vous pouvez définir l'attribut `Hint` d'un contrôle, et définir comme `True` l'attribut `ShowHint` (montre conseil) pour faire apparaître le conseil lorsque l'utilisateur place le pointeur de la souris sur un contrôle et l'y laisse un moment.

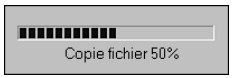
Figure 7.23
Un exemple de bulle d'aide.



Indicateurs de progression

Dans Windows 3.1, lorsque votre application effectuait une longue opération, vous utilisiez l'icône de sablier pour faire patienter l'utilisateur. Avec Windows 95, vous disposez d'une solution plus élégante et plus informative. L'indicateur de progression indique à l'utilisateur que quelque chose est en train de se passer, et l'informe de l'état d'avancement approximatif de cette opération. Un utilisateur est bien plus patient quand il sait que quelque chose de constructif est en cours.

Figure 7.24
Un exemple de barre de progression.



Barres d'outils

Les barres d'outils améliorent spectaculairement la productivité de l'utilisateur. Elles permettent un accès rapide aux opérations les plus souvent utilisées dans votre applications. On appelle parfois les barres d'outils palettes ou boîtes à outils. En Delphi 3, une barre d'outils est généralement implémentée à l'aide d'un contrôle panel (volet) et d'un nombre de contrôles de speed buttons. Ces boutons sont placés sur le volet et apparaissent habituellement au-dessous de la barre de menus. Si vous sélectionnez Fichier/Nouveau/Projets et application SDI ou MDI, vous disposez d'une barre d'outils dans le projet par défaut. Examinez attentivement les contrôles et ce qu'ils représentent.

Figure 7.25
Un exemple de barre d'outils.

Concevoir des écrans

Les images que nous présentons aux utilisateurs peuvent avoir de nombreux effets, de l'inspiration à la confusion. C'est pourquoi il est important de concevoir vos applications pour inspirer l'utilisateur, sans le distraire de son travail.

Pour concevoir vos écrans, vous pouvez vous appuyer sur l'expérience de nombreux éditeurs, Microsoft compris, qui ont mené beaucoup de recherche dans ce domaine. Dans la partie qui suit, nous examinerons ensemble les découvertes faites pas Microsoft. Toutes ces informations font partie des spécifications de Windows 95 ou se trouvent dans le CD-ROM du Microsoft Developer Network (MSDN) où vous pourrez puiser des idées qui vous aideront dans le développement.

Organisation

Il existe en fait six principes d'organisation que Microsoft a mis en valeur. Examinons-les un par un :

- Lisibilité et flux
- Structure et équilibre
- Relations entre éléments
- Thème fondateur et mise en évidence
- Hiérarchie de l'information
- Unité et intégration

Lisibilité et flux

Selon le principe de lisibilité et de flux, vous devez faire en sorte que votre conception communique vos idées de manière simple et directe, en évitant dans la mesure du possible les interférences visuelles. Pour réduire au maximum les interférences visuelles et améliorer la lisibilité et le flux, vous devez vous posez les questions suivantes lorsque vous concevez une boîte de dialogue ou une fenêtre :

- L'idée est-elle présentée de la façon la plus simple possible ?
- L'utilisateur peut-il parcourir facilement cette boîte de dialogue telle qu'elle est conçue ?
- Tous les éléments de cette fenêtre ont-ils une bonne raison d'y figurer ?

Structure et équilibre

Ce principe de structure et d'équilibre repose sur l'idée que sans une fondation solide, tout bâtiment est voué à la ruine. La structure de votre application c'est la façon dont votre application

— dans un ensemble — est mise en place. Sans bonne structure, l'ordre et le sens sont absents. Les relations entre les écrans et les informations qui figurent sur ces écrans jouent un rôle prépondérant dans la façon dont l'utilisateur perçoit votre application. Quant à l'équilibre, il s'agit de la manière dont sont réparties vos informations sur les écrans. Si un écran contient trop d'informations et qu'un autre n'en contient pratiquement pas, votre application semblera déséquilibrée. Si votre application manque de structure et d'équilibre, l'utilisateur éprouvera plus de difficultés à comprendre votre interface.

Les relations entre éléments

Il est important que vous indiquiez visuellement quelles sont les relations existant entre les différents éléments de votre application. Si un bouton élargit le champ des informations dans une boîte de liste, il est important que ces deux éléments soient connectés visuellement pour que leur relation apparaisse évidente aux yeux de l'utilisateur. Vous pouvez donc dans ce cas placer le bouton et la boîte de liste très près l'un de l'autre, ou les placer tous deux sur le même contrôle biseau. Si votre écran n'est qu'un amas anarchique de boutons, personne ne pourra — ni ne voudra — jamais l'utiliser.

Thème fondateur et mise en évidence

Vous devez identifier clairement le thème ou l'idée centrale à votre écran. Le concept de mise en évidence consiste à choisir le thème principal et à le rendre le plus explicite possible pour que l'utilisateur comprenne quelles choses sont les plus importantes dans l'écran. Ces deux concepts renvoient également aux concepts de structure et d'équilibre. Si votre application a un thème fondamental clair, la structure de l'application apparaîtra clairement à l'utilisateur. Votre application doit mettre l'accent sur l'accomplissement d'une tâche, et sur un accomplissement réussi. Un exemple d'application au thème fondamental clair est WinZip. Ce programme se consacre exclusivement à la compression et à la décompression de fichiers. Il ne comporte pas de défragmenteur de disque, ni d'Explorateur ni d'application de gestion de compte bancaire. Il met l'accent sur la compression des fichiers et s'acquitte parfaitement de cette tâche. L'interface est concise et pertinente, et permet d'arriver rapidement au résultat voulu.

Hiérarchie de l'information

Le concept de hiérarchie s'applique à la conception d'écrans mais aussi aux données (comme dans le contrôle d'Affichage d'arborescence). Vous devez déterminer quelles sont les informations les plus importantes pour les mettre sur l'écran initial, puis décider de ce qui figurera sur le deuxième écran, et ainsi de suite. Vous devez vous poser de nombreuses questions pour vous aider à construire la structure de votre hiérarchie : Quelles sont les informations les plus importantes pour l'utilisateur ? Que doit faire l'utilisateur en tout premier lieu ?... Que doit voir l'utilisateur sur le premier écran ? Sur le deuxième ? Vous devez également deviner quelles seront les priorités de l'utilisateur. L'organisation de votre écran facilitera-t-elle la tâche de l'utilisateur ou la compliquera-t-elle ? Sur quelles parties faut-il mettre l'accent ?

Unité et intégration

Nous passons maintenant à la vue d'ensemble. Comment votre application s'insère-t-elle dans l'ensemble du bureau et comment interagit-elle avec d'autres applications ? Si votre application se mure dans l'autisme, elle sera bien inutile.

Couleurs et palettes

Comme nous l'avons indiqué au début de cette leçon, vous pouvez faire revêtir à votre application l'ensemble des couleurs disponibles dans Windows. Les couleurs tiennent lieu d'indices visuels et permettent également d'attirer l'attention sur certaines parties de l'écran. Par ailleurs, on a tendance à associer des couleurs à différents états.

Sans réflexion sur l'emploi des couleurs, il est possible de s'attirer bien des désagréments. Un trop grand nombre de couleurs (ou des couleurs non adéquates) peuvent détourner l'attention de l'utilisateur ou le dérouter. Voici quelques remarques à prendre en considération lorsque vous choisissez les couleurs de votre application :

- L'association de couleurs à des significations précises n'est pas forcément évidente pour un utilisateur. Si en Europe le signe Stop est associé au rouge, ce n'est pas nécessairement le cas partout.
- Une autre personne n'a pas forcément les mêmes goûts que vous en matière de couleur. Il est donc préférable d'utiliser autant que possible les couleurs système. L'utilisateur pourra modifier la palette de couleurs Windows et votre application suivra.
- Il est possible que quelqu'un souhaite un jour utiliser votre application sur un moniteur monochrome.
- Les couleurs peuvent prendre différentes significations selon les pays. Il serait dommage de créer un superbe utilitaire pour s'apercevoir que la moitié du monde refuse de l'utiliser parce que les couleurs qu'il utilise ont une connotation négative.
- Microsoft a estimé que neuf pour cent de la population adulte mâle a des difficultés à bien distinguer les couleurs.

Les couleurs doivent être utilisées pour ajouter à l'affichage, et pas comme moyen principal de présenter l'information. Les formes, les motifs et d'autres concepts peuvent être utilisés pour que l'utilisateur distingue les différents types d'informations sur vos écrans. Microsoft suggère même de construire votre écran en noir et blanc et de n'ajouter la couleur qu'après.

Utiliser un nombre limité de couleurs signifie également choisir les bonnes combinaisons de couleur. L'usage de couleurs vives, telles que le rouge, sur un fond noir ou vert rend vos écrans peu lisibles. Il n'est pas recommandé d'employer des couleurs trop opposées. Une couleur neutre, gris clair par exemple, fait souvent le meilleur fond (comme le montrent la plupart des produits Microsoft). N'oubliez pas non plus que les couleurs claires ont tendance à ressortir tandis que les couleurs sombres semblent s'enfoncer.

Disposition

L'une des remarques les plus intéressantes dans les spécifications Windows est celle qui affirme que le but est de concevoir un environnement prévisible. C'est ce qu'aime un utilisateur, pouvoir deviner l'emplacement d'un élément de menu (ainsi, A propos est toujours le dernier élément du menu Aide). Vous devez toujours garder cette remarque à l'esprit. L'utilisation de votre application ne doit pas être une corvée.

L'espacement, l'utilisation des polices et la disposition à l'écran des contrôles et des informations garantissent le succès ou l'échec de votre application. L'utilisation que vous faites des polices est à cet égard d'une importance primordiale. Certaines polices sont préférables à d'autres. MS San Serif corps 8 est la police choisie par Microsoft pour tous les éléments système. Il convient de prendre d'autres facteurs en considération. Une police mise en italique est plus difficile à déchiffrer : les bordures sont plus irrégulières que dans une police normale. Il est préférable d'utiliser la police système par défaut dans la mesure du possible.

L'utilisateur peut toujours modifier la police par défaut, vous ne devez donc pas partir du principe que MS San Serif sera toujours la police système. Votre application doit pouvoir s'accommoder de nouvelles polices. Enfin, il faut savoir que les polices seront toujours moins belles à l'écran qu'une fois imprimées, à plus forte raison sur des moniteurs de basse résolution.

Unité de mesure

Dans ses spécifications, Microsoft utilise les unités de base des boîtes de dialogue comme unités de mesure. Ce système a été adopté car il est indépendant du matériel et se fonde sur la taille de la police système par défaut. Cela compense les variations de la taille de l'écran sur différents systèmes. Il existe un appel de fonction `GetDialogBaseUnits()` dans l'API Win32 qui renvoie une valeurs 32 bits contenant les unités de base de dialogue, basées sur la police système actuelle. Le mot de poids faible de la valeur de retour contient l'unité de base horizontale de boîte de dialogue, et le mot de poids fort l'unité de base de boîte de dialogue verticale.

Note

Ces considérations sur les unités de base de dialogue visent à vous faire comprendre les préoccupations de Microsoft. Delphi 3 utilise des unités liées au matériel (des pixels) pour l'espacement et l'alignement, et pas des unités de base de dialogue. Les mesures en pixels sont plus faciles à comprendre et à manipuler pour le programmeur.

La base horizontale est égale à la largeur moyenne en pixels des caractères ["a".."z","A".."Z"] dans la police système courante. L'unité de base verticale est égale à la hauteur en pixels de la police. Chaque unité de base horizontale est égale à quatre unités de dialogue horizontales, et chaque unité de base verticale est égale à huit unités de dialogue verticales. Par conséquent, pour convertir des unités de dialogue en pixels, votre application doit utiliser les formules suivantes :

```
PixelsEnX = (UnitesDialogueX x UniteBaseX) / 4
PixelsEnY = (UnitesDialogueY x UniteBaseY) / 8
```

Et pour convertir des pixels en unités de dialogue, les formules sont les suivantes :

```
UnitesDialogueX = (PixelsenX x 4) / UnitesBaseX
UnitesDialogueY = (PixelsEnY x 8) / UnitesBaseY
```

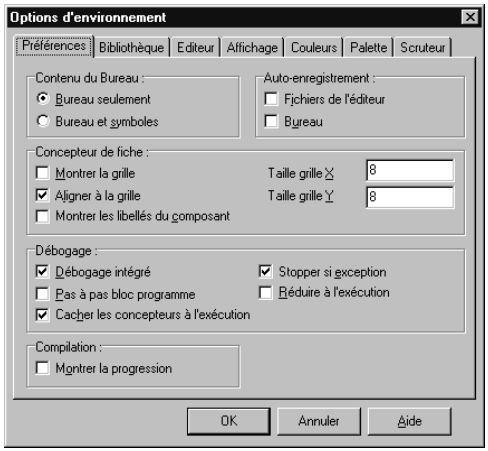
La multiplication est effectuée avant la division pour éviter des problèmes d'arrondi si les unités de base ne sont pas divisibles par quatre ou par huit. `PixelsEnX` et `PixelsEnY` vous permettent de savoir quel sera le multiplicateur à utiliser pour l'espacement, et ainsi de suite. Il existe des recommandations concernant la taille des différents éléments. Les boîtes d'édition, les libellés, les compteurs et les boutons doivent faire 14 unités dialogue de hauteur. Cette hauteur vous garantit un espace parfait au-dessus et au-dessous du lettrage.

Regrouper et espacer les éléments

Lorsque vous développez vos écrans, il est important que vous espaciez correctement les éléments de votre écran. Il est également important de conserver une marge constante (de sept unités dialogue) sur le pourtour de la fenêtre. Les contrôles doivent être espacés au minimum de quatre unités dialogue. L'exception à cette règle se rencontre lorsque vous souhaitez regrouper des boutons de barre d'outils. Dans ce cas, des boutons de fonctions voisines peuvent être directement adjacents.

Vous devez toujours regrouper les éléments de fonctions voisines. Le contrôle boîte de groupe est idéal pour cela, même si l'espacement seul peut suffire. Les boîtes de groupe permettent d'attirer l'attention de l'utilisateur sur un ensemble d'éléments particuliers. Il n'est pas recommandé de regrouper les contrôles au moyen de couleurs (en faisant figurer une forme colorée derrière les contrôles par exemple). Cette méthode peut prêter à confusion, et si l'utilisateur change la palette de couleurs, le résultat peut devenir franchement laid.

Figure 7.26
Regrouper les contrôles.



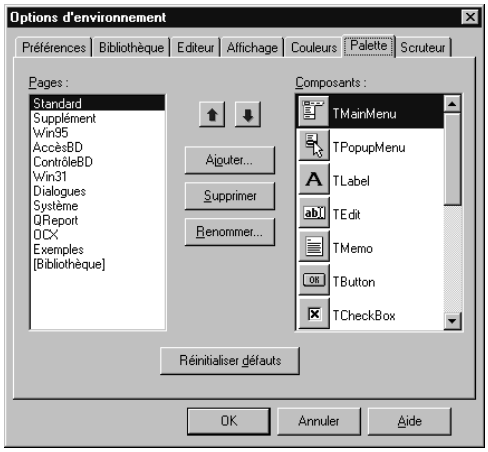
Aligner les éléments

Vous disposez de plusieurs méthodes pour disposer les données sur votre écran. Dans les pays occidentaux, nous lisons généralement de gauche à droite et de haut en bas. Les données les plus importantes se trouvent généralement dans le coin supérieur droit. Lorsque les données (ou les éléments) sont disposées verticalement, les côtés gauches de ces éléments doivent être alignés. Si vous associez des libellés aux contrôles, ces libellés doivent être placés au-dessus ou à gauche des contrôles, et doivent aussi être alignés à gauche. Cela s'applique à des contrôles tels que les contrôles d'édition, de boîte de liste et de boîtes à options. L'exception se rencontre avec les contrôles Boîte radio et case à cocher. Ces contrôles sont simplement alignés à gauche, avec leurs libellés disposés à leur droite.

La palette d'alignement permet d'aligner facilement les contrôles dans Delphi 3. Vous sélectionnez les contrôles à aligner, puis choisissez l'option d'alignement dans la palette d'alignement et vous pouvez ainsi rapidement placer les contrôles dans la disposition souhaitée. La fonction Verrouiller contrôles qui se trouve dans le menu Edition empêche les contrôles de se déplacer une fois que vous les avez correctement alignés.

Lorsque vous placez des boutons de commande dans une fenêtre, si le bouton de commande se trouve dans une boîte de groupe, on suppose de façon implicite que ce bouton n'affecte que les informations du groupe. Si le bouton de commande apparaît en dehors des boîtes de groupe, il est censé affecter l'ensemble de la fenêtre.

Figure 7.27
Boutons de commande (locaux et globaux).



Les boîtes de dialogue à onglets

Les boîtes de dialogue à onglets sont extrêmement utiles et vous permettent de conserver une interface propre et simple. Lorsque vous créez des boîtes de dialogues à onglets, il est préférable que les onglets soient tous de même taille pour donner un aspect uniforme. Les onglets peu-

vent permettre un classement par fonction, chaque onglet représentant une information sur un sujet précis. Outils/Options dans Delphi 3 est un bon exemple de boîte de dialogue à onglets.

Boîtes de dialogue modales

Tout d'abord, nous devons définir ce qu'est une boîte de dialogue modale. Il s'agit d'une fenêtre qui apparaît pour fournir ou demander des informations. Les boîtes de dialogue modales peuvent afficher des informations supplémentaires au sujet de données qui se trouvent dans une fenêtre primaire. Elles ne peuvent pas cependant constituer la fenêtre primaire de votre application. Cette boîte de dialogue ne contient pas tous les éléments présents dans une fenêtre primaire. La liste ci-après indique les différences entre une fenêtre primaire et une boîte de dialogue modale :

- Une boîte de dialogue modale ne comporte pas de bouton Agrandissement ou réduction.
- Elle peut avoir un bouton Fermer pour faire disparaître la fenêtre (bien que ce ne soit pas obligatoire).
- Le texte de la barre de titre décrit la fonction de la fenêtre.
- La barre de titre ne comporte pas d'icône.
- Il n'y a pas de barre d'état.
- Il peut y avoir un bouton "Qu'est-ce que c'est ?" qui fournit de l'aide à l'utilisateur sur les composants de la fenêtre.

Lorsque vous affichez des boîtes de dialogue modales, il est recommandé qu'elles ne dépassent pas une taille de 263 × 263 unités de base de dialogue. Si une fenêtre secondaire est une feuille de propriétés, les tailles recommandées sont 218 × 227, 215 × 212 et 212 × 188 en unités de base de dialogue. On estime que ces tailles permettent d'offrir une surface appréciable sans prendre pour autant trop de place (particulièrement dans les résolutions les plus basses).

Comportement des fenêtres

Vos boîtes de dialogue doivent apprendre à bien se comporter dans un environnement à multiples fenêtres. Il existe un comportement standard pour les boîtes de dialogue. Elles peuvent apparaître en deux occasions. La première, lorsqu'une boîte de dialogue est créée en réponse à une commande choisie dans la fenêtre primaire. C'est le cas de la boîte de dialogue Rechercher qui apparaît dans Microsoft Word 7. Dans ce cas, lorsque vous minimisez la fenêtre primaire, toutes les boîtes de dialogue qui lui sont associées doivent être minimisées elles aussi. Lorsque la fenêtre primaire est restaurée, les boîtes de dialogue reviennent en ordre à leur position en Z.

Nouveau

L'ordre en Z décrit les relations de couche entre différents objets tels que des fenêtres. L'ordre en Z dans Windows est l'ordre dans lequel les fenêtres sont dessinées à l'écran. La fenêtre au sommet de l'ordre en Z est également la fenêtre qui apparaîtra au-dessus de toutes les autres fenêtres sur votre écran.

Un deuxième exemple de boîte de dialogue est celle qui est générée par le système, ou en dehors de votre application. C'est le cas de la fenêtre Propriétés de l'affichage de Windows 95. Dans ce cas, la fenêtre de propriétés doit rester ouverte lorsque vous minimisez votre fenêtre d'application. Ces deux comportements se retrouvent dans l'environnement Delphi. Lorsque vous minimisez votre application Delphi, toutes les fenêtres enfant (pour une application MDI) et secondaires sont elles aussi minimisées.

Les boîtes de dialogue ne doivent pas apparaître sur la barre des tâches de Windows 95. Seules doivent y apparaître les fenêtres principales. Les applications Delphi que vous générez se chargent de faire respecter ces règles. Si vous ajoutez une boîte de dialogue à votre projet et définissez `sStayOnTop` comme valeur de l'attribut `FormStyle`, vous pouvez minimiser la fenêtre principale, la fenêtre secondaire l'imitera alors mais seule la fenêtre principale apparaîtra sur la barre des tâches. Les fenêtres qui restent toujours en avant-plan peuvent être très énervantes, faites-en donc un usage modéré.

Si vous employez des fenêtres en cascade dans votre application (une fenêtre secondaire ouvrant une autre fenêtre secondaire), ayez ces deux points à l'esprit. Tout d'abord, limitez le nombre à trois fenêtres (la principale, la secondaire et la sous-secondaire). Ne descendez pas plus bas que cela, sinon vous risquez de dérouter l'utilisateur. Ensuite, assurez-vous lorsque vous ouvrez les fenêtres secondaire et sous-secondaire qu'elles apparaissent décalées par rapport à la fenêtre principale, vers le bas et la droite. Cela donne un effet de cascade aux fenêtres et indique à l'utilisateur qu'il descend dans les couches de fenêtres.

Nous avons parlé un peu plus tôt des fenêtres dépliantes. Dans ce cas, dans une fenêtre secondaire, vous appuyez sur un bouton qui comporte un >> après son libellé et la fenêtre s'élargit pour révéler des informations supplémentaires. Cette méthode s'avère très utile pour révéler des informations par paliers successifs. Si l'utilisateur s'estime satisfait par le niveau de détail de la page la plus petite, il peut en rester là. Sinon, il peut toujours étendre la fenêtre.

Vous pouvez également ouvrir des fenêtres secondaires sous deux modes : modal et non modal. Si vous ouvrez les fenêtres en mode modal, l'utilisateur peut toujours accéder aux autres fenêtres de votre application, sans être obligé de fermer votre fenêtre au préalable. Si vous ouvrez une fenêtre en mode non modal, l'utilisateur doit fermer votre fenêtre avant de pouvoir accéder aux autres fenêtres de votre application. C'est une des rares occasions où vous forcez l'utilisateur à faire quelque chose (entrer un mot de passe avant de pouvoir continuer par exemple), et vous ne devez pas oublier à quel point les gens détestent recevoir des ordres. Utilisez les boîtes de dialogue et les fenêtres non modales avec parcimonie, elles ont tendance à ralentir la tâche de l'utilisateur.

Récapitulatif

Lors de cette journée, nous avons vu ensemble les concepts à l'œuvre dans la conception d'une interface graphique utilisateur. Nous avons examiné les différents moyens de donner le contrôle à l'utilisateur (ou du moins de lui donner cette impression) ainsi que les différentes techni-

ques de conception assurant que votre écran reste dégagé et lisible. Bien que nous n'ayons pu qu'effleurer le sujet, vous avez sans doute pu vous faire une petite idée de ce qu'implique la conception d'une bonne interface. Après avoir vu les contraintes qui pèsent sur un concepteur de GUI, on ne peux s'empêcher de considérer avec un respect renouvelé des applications telles que Delphi 3. Bonne chance pour votre projet d'GUI !

Atelier

L'atelier vous donne trois façons de vérifier que vous avez correctement assimilé le contenu de cette leçon. La section Questions - Réponses vous donne des questions qu'on pose couramment, ainsi que leur réponse, la section Questionnaire vous pose des questions, dont vous trouverez les réponses en consultant l'Annexe A, et les Exercices vous permettent de mettre en pratique ce que vous venez d'apprendre. Tentez dans la mesure du possible de vous appliquer à chacune des trois sections avant de passer à la suite.

Questions - Réponses

Q Nous voulons écrire une application et nous pensons qu'elle ferait une excellente application MDI. Comment en être sûr ?

R C'est une bonne question. Comme nous l'avons dit au cours de cette leçon, Microsoft n'encourage pas la création d'applications MDI, car peu de gens savent en écrire correctement. Si vous pensez que vous devez le faire, n'oubliez pas que toutes les fenêtres filles doivent être identiques. Il ne faut pas mélanger les types de documents dans les fenêtres filles. Vous devez également laisser le contrôle des fenêtres à l'utilisateur.

Q Où trouver des informations supplémentaires ?

R Vous pouvez consulter le CD du Microsoft Developer Network, qui contient quantité d'informations sur les spécifications du logo Win95 et sur la conception d'une GUI en général.

Questionnaire

1. Quelle est la fonction principale du bouton droit de la souris dans Windows 95 ?
2. Pourquoi Microsoft recommande-t-il des tailles bien précises pour les fenêtres secondaires ?
3. Pourquoi est-il important que votre application reprenne la disposition standard des menus ?

Exercices

1. Allez dans la palette de couleurs de Delphi 3 et essayez de comprendre à quoi servent toutes les couleurs système. Si vous avez besoin d'aide, placez différents types de contrôle sur une fiche et regardez les couleurs qui sont données par défaut à chacun des éléments.
2. Ouvrez une application Windows95 classique et regardez comment les menus contextuels sont utilisés pour proposer des commandes à l'utilisateur.
3. Regardez les applications que vous utilisez quotidiennement. Satisfont-elles tous les standards évoqués dans cette section ?

8

La bibliothèque de composants visuels

Delphi propose une vaste bibliothèque de composants préconstruits. Dans les journées qui ont précédé, vous avez pu découvrir l'EDI de Delphi ainsi que les principes de la programmation orientée objet. Vous avez déjà pu utiliser des exemples utilisant la VCL (*Visual Component Library* ou Bibliothèque de composants visuels), mais pour l'instant, vous n'avez en fait qu'effleurer le sujet. Lors de cette journée, nous allons explorer la VCL. En examinant chaque onglet de la palette VCL, vous verrez quels sont les objets disponibles, et vous découvrirez les propriétés et Gestionnaires d'événements associés à ces objets. Puis nous construirons un programme de démonstration en mettant à contribution de nombreux composants. Nous parlerons brièvement de composants Internet, Database, Decision Cube, ActiveX et ReportSmith.

Qu'est-ce que la VCL ?

Nouveau

VCL est l'acronyme de Visual Component Library *(Bibliothèque de composants visuels). Cette bibliothèque est composée d'objets préconstruits appelés composants. Pour simplifier, ces composants sont à Delphi ce que les OCX et les contrôles ActiveX sont à VB. Cependant, à la différence des OCX, les composants de la VCL sont écrits en Pascal Objet Delphi et sont stockés dans une unique bibliothèque, au lieu de figurer dans des fichiers distincts. Tout composant que vous utilisez devient partie intégrante de votre exécutable.*

Ces composants vous épargnent bien des heures — voire des jours — de développement et de tests, en vous permettant de concevoir visuellement votre application. Vous sélectionnez des composants, les déposez sur une fiche et définissez quelques propriétés, et vous voilà avec une application créée sans pratiquement aucun code. La VCL est constituée de composants offrant les fonctionnalités usuelles que l'on peut trouver dans la plupart des applications Windows. Grâce au grand nombre de composants proposés par Delphi, vous pouvez vous consacrer à la création de votre application, sans perdre de temps à réinventer la roue pour créer votre interface graphique, vos fonctions de bases de données et les routines de communication de programme à programme (OLE et DDE). Grâce au Pascal Objet de Delphi vous pouvez créer très rapidement n'importe quelle application. Votre code ne fait que lier les différents composants. On peut comparer cela à la construction d'une maison préfabriquée, par opposition à la construction d'une maison en ne partant de rien. Si vous partez de rien, vous devez trouver de nombreuses planches, des clous, divers matériaux, etc. et vous devez tailler et raboter votre bois. En revanche, pour la maison préfabriquée, les murs et les diverses parties principales de la maison sont déjà assemblés, même s'ils ne sont pas encore réunis. Il suffit de les réunir et votre maison est vite bâtie. Elle n'est pas moins solide que la maison "artisanale", mais le constructeur n'a pas eu à se préoccuper de tout jusqu'au moindre détail.

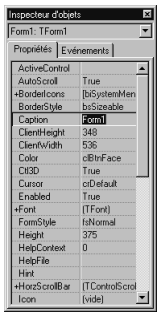
Vous pouvez créer en Delphi vos propres composants et les ajouter à la palette de la VCL pour un usage ultérieur. Lorsque vous consultez la VCL, les divers composants sont référencés par leur nom qui apparaît lorsque vous placez sur eux le pointeur de la souris (Button par exemple). Le nom des composants commence toujours par un T (TButton par exemple), nom sous lequel on les retrouve dans l'Inspecteur d'objets et dans l'aide en ligne. Il faut donc vous habituer à penser à `TForm` si vous voyez Form (fiche), à `TEdit` si vous voyez à Edit, etc.

Que se passe-t-il en fait lorsque vous déposez un composant sur une fiche ? Delphi génère automatiquement le code nécessaire à l'utilisation du composant et met à jour le projet en conséquence. Vous n'avez besoin que de définir les propriétés, placer du code dans les Gestionnaires d'événements et d'utiliser des méthodes de façon appropriée pour parvenir au but recherché.

Propriétés

Ces composants sont bien beaux, mais vous devez trouver un moyen de leur indiquer comment apparaître et se comporter. Les composants Delphi, tout comme les OCX de VB, ont des propriétés. Ces propriétés vous permettent de modifier la taille, la forme, la visibilité, la position d'un composant et autres caractéristiques. Pour accéder à ces propriétés, on utilise l'Inspecteur d'objets (voir Figure 8.1). Celui-ci comporte deux onglets : l'onglet Propriétés et l'onglet Evénements. Intéressons-nous pour l'instant à l'onglet Propriétés.

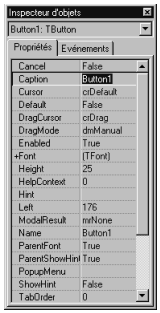
Figure 8.1
L'onglet Propriétés de l'Inspecteur d'objets.



La plupart des propriétés sont communes à tous les composants, mais pas toutes. Par ailleurs, certaines propriétés sont spécifiques à un composant ou à un groupe de composants. Il est impossible de décrire ici toutes les propriétés de tous les composants, nous allons donc nous intéresser aux propriétés les plus communes ainsi qu'aux types de propriétés et à la façon de les utiliser. Regardons les propriétés d'un composant que vous avez déjà eu l'occasion d'utiliser, le bouton (Button). Dans Delphi, ouvrez un nouveau projet. Allez dans l'onglet Standard de la palette des composants. Sélectionnez le bouton en cliquant dessus puis cliquez sur la fiche. Vous ajoutez ainsi un nouveau composant Button à votre fiche. La Figure 8.2 vous montre l'Inspecteur d'objets sur lequel est sélectionné l'onglet Propriétés.

Figure 8.2
La palette Standard, avec le composant Button sélectionné.

Figure 8.3
L'onglet Propriétés de l'Inspecteur d'objets pour le composant Button.



Comme vous pouvez le voir, de nombreuses propriétés peuvent être définies pour le composant Button. Certaines acceptent des valeurs numériques comme paramètres, par exemple la propriété `Height`. D'autres propriétés recèlent une boîte à options dans laquelle vous sélectionnez une liste de paramètres prédéfinis (ce sont des constantes définies dans Delphi), comme la propriété `Cursor`. D'autres encore vous laissent le choix entre True et False, comme c'est le cas pour la propriété `Default`. Ou bien, elles acceptent du texte normal, comme la propriété `Caption`. Enfin, certaines propriétés comportent des menus contextuels et des éditeurs vous permettant de procéder à vos réglages.

Il est un autre type de propriétés que vous devez connaître : la propriété imbriquée. Une propriété imbriquée est toujours précédée par un "+", comme c'est le cas pour la propriété +Font par exemple. Une propriété imbriquée est une propriété de propriété. Lorsque vous double-cliquez sur une propriété comportant des propriétés imbriquées, cette propriété se développe pour vous montrer les propriétés imbriquées. Le "+" se change alors en "–". Les propriétés peuvent être imbriquées sur plusieurs niveaux. Le "+" et le "–" sont utilisés de la même manière dans chaque niveau successif. Pour réduire une liste de propriétés imbriquées, il vous suffit de double-cliquer sur le "–", qui se transforme alors de nouveau en "+". Regardez la liste des propriétés du composant Button. Toutes les propriétés imbriquées ont été développées :

- `Cancel`
- `Caption`
- `Cursor`
- `Default`
- `DragCursor`
- `DragMode`
- `Enabled`
- `-Font`

```
    Color
    Height
    Name
    Pitch
    Size
-Style
        fsBold
        fsItalic
        fsUnderline
        fsStrikeOut
Height
HelpContex
Hint
Left
ModalResult
Name
ParentFont
ParentShowHint
PopupMenu
ShowHint
TabOrder
TabStop
Tag
Top
Visible
Width
```

En modifiant ces propriétés, vous pouvez contrôler la taille, l'apparence, la couleur, la police et bien d'autres choses encore. Une des propriétés les plus communément utilisées dans des composants tels que le bouton est la propriété `Caption` (titre ou libellé). Si vous créez un bouton OK, vous modifiez la propriété `Caption` de bouton pour qu'elle devienne OK. Le bouton affiche alors OK. De même, vous pouvez définir la propriété `Name` du bouton comme étant `MonBoutonOK` pour que le code de l'unité soit plus lisible.

Note *Ici aussi, les propriétés sont trop nombreuses pour que nous puissions toutes les examiner. En lisant cette leçon, et le reste du livre, vous découvrirez un bon nombre de propriétés. Lorsque vous avez besoin d'informations concernant une propriété particulière, il vous suffit de cliquer sur cette propriété et d'appuyer sur F1. L'aide en ligne explique chaque propriété en détail. Vous pouvez, dans un premier temps, vous astreindre à lire la description de plusieurs propriétés chaque jour. Et n'oubliez pas que les fiches ont elles aussi des propriétés.*

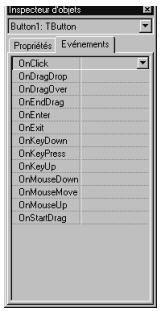
Il vous faut également savoir que certaines propriétés dépendent des paramètres d'autres propriétés. C'est le cas de la propriété `Hint`. Vous définissez `Hint` comme un message texte qui

s'affichera lorsque le pointeur de la souris sera laissé sur le contrôle pendant un certain temps. Ceci ne fonctionnera que si vous avez défini comme `True` (vraie) la propriété `ShowHint` du contrôle ou de la fiche ou du conteneur parent (GroupBox par exemple). En effet, la propriété `ParentShowHint` du contrôle est définie comme `True` par défaut, ce qui transmet le paramètre `False` aux contrôles enfants. On peut aussi citer le cas de la propriété `ParentFont`. Si `ParentFont` est définie comme `True`, le composant prend ses informations de police dans la fiche ou le conteneur parent. Ne soyez pas découragé devant le nombre de propriétés, car la plupart des paramètres par défaut pour ces propriétés conviennent parfaitement dans la plupart des cas. A force d'utiliser Delphi, ces propriétés iront de soi. Les noms sont relativement explicites (si vous avez quelques notions d'anglais) et si vous avez un problème, vous pouvez toujours appuyer sur la touche F1.

Evénements

De nombreux événements ou Gestionnaires d'événements sont associés aux composants Delphi. Lors de la journée précédente, nous avons vu que le code que vous stockez dans ces Gestionnaires d'événements est exécuté lorsque l'événement précis survient (si par exemple la souris se déplace, en cas de clic, etc.). Considérez l'onglet Evénements de l'Inspecteur d'objets pour le composant `Button` (voir Figure 8.3).

Figure 8.4
L'onglet Evénements de l'Inspecteur d'objets.



Pour créer un Gestionnaire d'événements, il suffit de double-cliquer sur l'événement que vous souhaitez créer. Vous vous retrouvez alors dans le gestionnaire et vous pouvez y entrer votre code. Si vous décidez finalement de ne pas utiliser le Gestionnaire d'événements, Delphi le supprime pour vous à la prochaine compilation, constatant que vous n'y ayez pas placé de code. Comme dans l'onglet Propriétés, vous pouvez cliquer sur un événement puis appuyer sur F1 pour lire l'aide en ligne concernant cet événement. Par ailleurs, lorsque vous cliquez à la droite d'un événement dans l'Inspecteur d'objets, vous faites apparaître une boîte à options. Il suffit alors de cliquer sur la flèche vers le bas pour voir une liste des événements actuellement définis et disponibles. Ceci vous permet de faire partager le même Gestionnaire d'événements

à plusieurs contrôles et/ou événements. Ainsi, vous n'avez pas besoin de reproduire plusieurs fois le même code.

Méthodes

Il existe plusieurs types de méthodes, mais nous allons nous intéresser aux méthodes associées aux divers composants, et à la façon dont ces méthodes les affectent. Les méthodes vous permettent elles aussi de contrôler les composants. Comme les composants sont des objets, on peut y déclarer des fonctions et des procédures. C'est la définition même d'une méthode : une fonction ou procédure publique, déclarée dans un objet, et que vous pouvez appeler. Prenons l'exemple de notre composant `Button`. Si vous voulez le rendre invisible, vous pouvez exécuter une ligne de code du type `Button1.Hide`, et le bouton deviendra invisible. Pour savoir quelles sont les méthodes disponibles pour un composant ou objet particulier, cliquez sur le composant et appuyez sur F1. Dans l'aide en ligne qui apparaît alors, cliquez sur Méthodes et vous pourrez voir la liste des méthodes disponibles. Vous pouvez ensuite cliquer sur la méthode qui vous intéresse. Voici des méthodes disponibles pour le composant `Button` :

- `BeginDrag`
- `BringToFront`
- `CanFocus`
- `ClientToScreen`
- `Create`
- `Destroy`
- `Dragging`
- `EndDrag`
- `Focused`
- `Free`
- `GetTextBuf`
- `GetTextLen`
- `Hide`
- `Refresh`
- `Repaint`
- `ScaleBy`
- `ScreenToClient`
- `ScrollBy`
- `SendToBack`
- `SetBounds`
- `SetFocus`
- `SetTextBuf`
- `Show`
- `Update`

De même que pour les propriétés, les noms sont assez parlants et vous donnent une idée de la fonction de la méthode. Cependant, à la différence des propriétés, les méthodes ne sont pas accessibles dans l'Inspecteur d'objets. Pour en savoir plus sur les méthodes, vous devez vous reporter à l'aide en ligne. Là encore, il n'est pas inutile de lire un peu d'aide en ligne chaque jour. Vous n'avez pas besoin d'apprendre tout par cœur, mais lorsque vous saurez ce qui est disponible, vous pourrez créer plus facilement vos applications. Il serait dommage de réinventer une fonction parce que vous ne saviez pas qu'il existait déjà une méthode effectuant la tâche souhaitée.

Composants visuels et non visuels

Il existe deux types de composants dans Delphi : les composants visuels et les composants non visuels. Les composants visuels sont ceux avec lesquels vous construisez votre interface utilisateur. Les composants des deux types apparaissent lors de la conception, mais les composants non visuels restent invisibles lors de l'exécution. `Button`, `Edit` et `Memo` sont des exemples de composants visuels. On utilise des composants visuels à des fins très diverses. `Timer`, `OpenDialog` et `MainMenu` sont des exemples de composants non-visuels. `Timer` permet au programmeur d'activer un code spécifique à des intervalles de temps donnés, et n'est jamais vu par l'utilisateur. `OpenDialog` et `MainMenu` sont des composants non visuels qui finissent par produire des résultats visibles et utilisables à l'écran, mais les composants eux-mêmes ne sont pas visibles. Les composants non visuels sont faciles à repérer car ils ne peuvent pas être redimensionnés et sont en tout point similaires au bouton qui vous a permis de les sélectionner. Tous les composants, visibles ou non, sont très utiles et vous font gagner un temps précieux.

Bibliothèque

La bibliothèque est subdivisée en treize onglets regroupés logiquement, parmi lesquels un onglet ActiveX et un onglet Exemples. Les onglets supplémentaires contiennent des exemples de contrôles et de composants. Il s'agit ici des onglets de la version CS de Delphi 3. Certains composants ne sont pas disponibles dans les autres versions.

- Standard
- Supplément
- Win32
- Système
- Internet
- AccèsBD
- ContrôleBD

- Decision Cube
- Qreport
- Dialogues
- Win31
- Exemples
- ActiveX

Même s'il est impossible de décrire et d'utiliser tous les composants et leurs propriétés, nous allons voir tous les composants et examinez certaines de leurs propriétés. Nous allons construire des exemples de fiches pour certains des onglets. L'aide en ligne est une aide précieuse en la matière, vous permettant de déterminer l'utilité d'un composant (cliquez sur le composant, puis appuyez sur F1 pour accéder à l'aide). Regardez chacun des onglets et construisez une application utilisant les composants qui y figurent. Cette application n'a pas de but précis, sinon de vous aider à comprendre comment fonctionnent les divers composants. Ouvrez un nouveau projet et enregistrez UNIT1.PAS sous STANDARD.PAS. Enregistrer le projet sous VCLDEMO.DPR. Allez dans l'onglet Standard et commencez l'application VCLDEMO.

Onglet Standard

Dans cet onglet se trouvent les composants les plus usuels. Il est visible par défaut au démarrage de Delphi et compte 14 composants, comme indiqué Tableau 8.1. Regardez-les rapidement avant de construire la première fiche de notre application d'exemple en utilisant certains d'entre eux (voir Figure 8.4). Remarquez que selon la résolution de votre écran, il est possible que vous ne puissiez pas voir tous les onglets. Dans ce cas, Delphi ajoute au bout de la barre d'onglets des flèches vous permettant de faire défiler les onglets, comme indiqué Figure 8.5.

Figure 8.5
Cliquez sur cette flèche pour voir plus de composants. L'onglet Standard de la palette de composants.

Tableau 8.1 : L'onglet Standard de la palette de composants

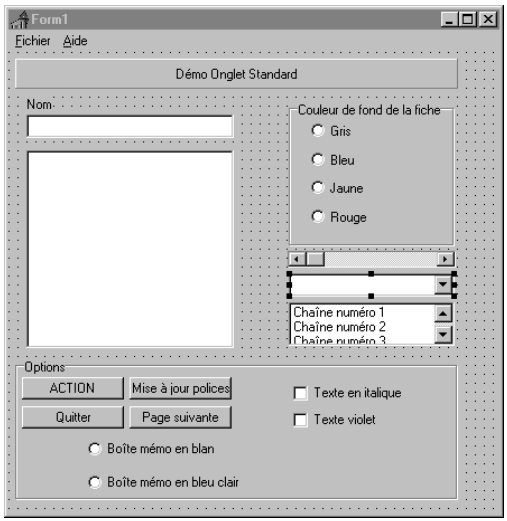
Composant	Description
	Le composant `MainMenu` (menu principal) vous permet de concevoir et de créer la barre de menus et les menus déroulants de votre fiche.

Composant	Description
	Le composant `PopupMenu` (menu contextuel) vous permet de concevoir et de créer des menus contextuels qui apparaissent lorsque l'utilisateur fait un clic droit.
	Le composant `Label` (libellé) permet de placer du texte dans une fiche ou dans d'autres conteneurs. Ce texte ne peut être modifié par l'utilisateur.
	Le composant `Edit` (édition) permet à l'utilisateur d'entrer une ligne de texte. Ce composant permet aussi d'afficher du texte.
	Le composant `Memo` permet d'entrer et/ou d'afficher plusieurs lignes de texte.
	Le composant `Button` (bouton) permet de créer des boutons que l'utilisateur utilisera pour sélectionner des options dans une application.
	Le composant `CheckBox` (case à cocher) permet à l'utilisateur de sélectionner ou désélectionner des options en cliquant dessus.
	Le composant `RadioButton` (bouton radio) permet d'offrir un ensemble d'options à l'utilisateur, au sein duquel il ne peut choisir qu'une option. Un ensemble est constitué d'un nombre quelconque de boutons radio placés dans un conteneur tel qu'une fiche, un volet, etc.
	Le composant `ListBox` (zone de liste) est la boîte de liste Windows standard qui permet de créer une liste d'éléments parmi lesquels l'utilisateur peut faire son choix.
	Le composant `ComboBox` (zone d'options) est semblable à une `ListBox` qui bénéficierait des avantages du composant Edit. Ce composant permet à l'utilisateur de sélectionner un élément d'une liste ou de l'entrer sous forme de texte.
	Le composant `ScrollBar` (barre de défilement) est la barre de défilement Windows standard qui permet de faire défiler des fiches ou des contrôles.
	Le composant `GroupBox` (zone de groupe) est un conteneur permettant de regrouper des éléments de même nature, tels que des `RadioButton`,`CheckBox`, etc.

Composant	Description
	Le composant `RadioGroup` (groupe radio) est un mélange de `GroupBox` et de `RadioButton` conçu spécifiquement pour créer des groupes de boutons radio. On peut y placer plusieurs boutons radio, mais pas de composants d'autres types.
	Le composant `Panel` (volet) est un conteneur permettant de regrouper d'autres conteneurs ou des composants. Les volets permettent également de construire des barres d'état, des barres d'outils et des palettes d'outils.

Maintenant que nous avons décrit tous les composants figurant dans cet onglet, commençons à travailler sur notre application. Dans le projet VCLDEMO.DPR que vous avez créé précédemment, modifiez le titre de la fenêtre (caption) pour qu'il devienne "L'onglet standard". Nous allons créer une fiche assez surchargée mettant à contribution au moins un composant de chaque type. Regardez la Figure 8.6 et servez-vous en comme modèle pour ajouter vos composants à la fiche, de manière à respecter approximativement les positions et les proportions que vous pouvez y voir.

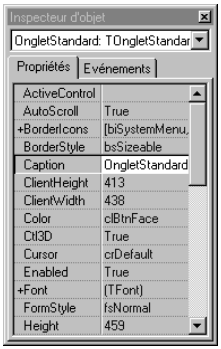
Figure 8.6
La fiche Onglet standard de VCLDEMO.



Note *Dans le reste du texte, nous utiliserons le nom anglais des composants et de leurs propriétés. Il vous suffit de vous référer à la liste précédente (ou à l'aide en ligne de Delphi) pour trouver l'équivalent français.*

Commencez par modifier la propriété `Name` en Onglet standard. Ensuite, ajoutez un composant `Panel` au sommet de l'écran. Etirez-le pour qu'il s'étende sur l'écran. Ce composant vous permettra de donner un titre à votre fiche. Définissez la propriété `Caption` de ce composant comme "Démo de l'onglet Standard". Le `Panel` est très utile pour créer des titres dotés d'un effet de relief, mais sert aussi de conteneur pour d'autres composants. Maintenant, ajoutez un label et donnez-lui comme `caption Nom`. Le composant label a une propriété appelée `AutoSize` (taille auto) qui est définie comme `True` par défaut. `Autosize` fait que le composant augmente ou diminue de taille lorsque vous modifiez son titre, la taille de la police, etc. Si la couleur de fond du label est la même que celle de la fiche (c'est le cas par défaut), vous risquez de perdre de vue votre label s'il rétrécit au point de ne contenir aucun caractère et qu'il n'est pas sélectionné. Dans ce cas, vous pouvez le retrouver facilement en allant dans l'Inspecteur d'objets et en utilisant la zone d'options pour dérouler la liste des composants dans laquelle vous pouvez sélectionner votre label (voir Figure 8.7). Entrez maintenant le texte désiré dans la propriété `Caption`.

Figure 8.7
L'Inspecteur d'objets pour le composant Name.



Ajoutez un composant `Edit` sous le label. Utilisez l'Inspecteur d'objets pour supprimer tous les caractères qui figurent dans la propriété `Text` du composant. Ajoutez un `MainMenu` et un `PopUpMenu`. N'oubliez pas que ce sont des composants non visuels et que l'endroit où vous les placez n'a donc pas d'importance. Le plus simple consiste encore à les mettre dans un coin à part pour qu'ils ne vous gênent pas lorsque vous placez vos autres composants. Ajoutez un composant `Memo`.

Prenez une minute pour regarder les propriétés disponibles avec le composant `Memo`. Utilisez l'aide en ligne pour voir la description de certaines de ces propriétés. Le composant `Memo` est un composant très puissant qui vous permet de concevoir votre propre éditeur de texte. Vous n'avez pas besoin ici d'utiliser toutes les propriétés de ce composant, mais il n'est pas inutile que vous ayez conscience de leur existence. Dans l'Inspecteur d'objets, double-cliquez sur la propriété `Lines` (lignes) et effacez le texte "Memo1". Appuyez quatre fois sur Entrée. Vous augmentez ainsi la taille du tampon du composant `Memo` (vous avez besoin de disposer d'au moins trois ou quatre lignes pour cet exemple, et vous les créez ainsi). Si vous ne faites pas cela, l'exemple ne fonctionnera pas. Ajoutez maintenant un `RadioGroup`, une `ScrollBar`, une

ComboBox et une ListBox à la fiche. Assurez-vous que tous les composants que vous avez placés ressemblent à ceux de la Figure 8.5.

Double-cliquez sur le composant RadioGroup que vous avez placé sur la fiche et ajoutez le code ci-après dans son événement OnClick :

```
procedure TStandardTab.RadioGroup1Click(Sender: TObject);
begin
     If RadioGroup1.ItemIndex=0 Then StandardTab.Color:=clSilver;
     If RadioGroup1.ItemIndex=1 Then StandardTab.Color:=clBlue;
     If RadioGroup1.ItemIndex=2 Then StandardTab.Color:=clYellow;
     If RadioGroup1.ItemIndex=3 Then StandardTab.Color:=clRed;
end;
```

Double-cliquez sur le composant ScrollBar et ajoutez le code ci-après dans l'événement OnChange :

```
procedure TStandardTab.ScrollBar1Change(Sender: TObject);
begin

     RadioGroup1.ItemIndex := ScrollBar1.Position;end;
```

Placez une GroupBox sur la fiche et assurez-vous qu'elle dispose d'assez de place pour contenir quelques composants. Ajoutez-y deux boutons, deux boutons radio et deux cases à cocher. Mettez Polices pour le Button2, Quitter pour le Button3 et Page suivante pour le Button4. Voilà, vous avez placé tous les composants de cette fiche.

Ajustez la taille et la position des composants jusqu'à ce que votre fiche ressemble à celle de la Figure 8.6. Il faut maintenant ajouter du code à ces boutons. Voici le code pour l'événement OnClick du Button1 :

```
procedure TStandardTab.Button1Click(Sender: TObject);
Var
   x : Integer;
begin
     {Efface le contenu actuel de TMemo}
     Memo1.Clear;
     {Copie le texte entré dans la boîte Nom (Edit1) dans le Memo}
     Memo1.Lines.Add(Edit1.Text);
     {Copie le texte entré dans la boîte de texte de la Combobox dans le
     Memo}
     Memo1.Lines.Add(ComboBox1.Text);
     {Copie le texte sélectionné de la boîte de liste vers le memo}
     Memo1.Lines.Add('ListBox String
        #' +IntToStr(ListBox1.ItemIndex+1));
```

```
     If RadioButton1.Checked then Memo1.Color:=clWhite;
     If RadioButton2.Checked then Memo1.Color:=ClAqua;
end;
```

Et le code de l'événement OnClick du Button2 :

```
procedure TStandardTab.Button2Click(Sender: TObject);
begin
     If CheckBox1.State = cbChecked then
        StandardTab.Font.Style:=[fsItalic]
     else StandardTab.Font.Style:=[];
     If CheckBox2.State = cbChecked then
        StandardTab.Font.Color:=clPurple
     else StandardTab.Font.Color:=clBlack;
end;
```

Et le code pour l'événement OnClick du Button3 :

```
procedure TStandardTab.Button3Click(Sender: TObject);
begin
     Close;
end;
```

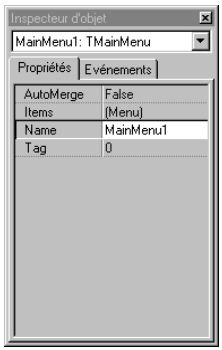
N'insérez pas tout de suite le code du `Button4`, qui permet de cacher cette fiche et d'afficher la suivante. Il sera ajouté dès que nous aurons créé une fiche pour les onglets suivants.

Note

N'oubliez pas que lorsque vous double-cliquez sur un composant, vous créez la déclaration de procédure comportant les mots clé `begin` et `end`. Il ne vous reste plus qu'à remplir le code figurant ci-avant. De plus, vous ne verrez pas le code correspondant à tous les contrôles de la fiche. En effet, Delphi nettoie lors de la compilation les Gestionnaires d'événements vides qui sont créés lorsque vous double-cliquez sur un composant. En fait, il est préférable de laisser Delphi se charger de ces suppressions, car en vous en chargeant vous-même, vous risquez de commettre des erreurs.

Enfin, ajoutez le code pour les composants `MainMenu` et `PopupMenu`. Regardez le composant `MainMenu`. A première vue, ce composant et son utilisation peuvent vous sembler étranges, tout particulièrement si vous avez déjà programmé en Visual Basic. Vous vous demandez peut-être où se trouve le Concepteur de menu. Vous pouvez l'activer par le biais de la propriété Items qui figure dans l'Inspecteur d'objets.

Figure 8.8
L'Inspecteur d'objets pour le composant MainMenu.



Si vous double-cliquez sur la valeur de la propriété `Items`, là où figure (Menu), vous voyez apparaître le Concepteur de menu (voir Figure 8.9).Vous pouvez également cliquer sur le composant `MainMenu` de votre fiche et utiliser le menu contextuel en cliquant à droite.

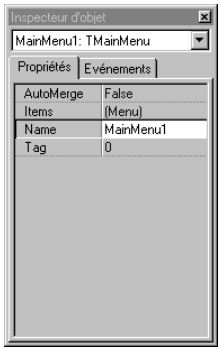
Figure 8.9
Le Concepteur de menu.

L'Inspecteur d'objets se modifie pour laisser de la place au Concepteur de menu (voir Figure 8.10). Là, vous pouvez modifier les titres des éléments de menu que vous souhaitez ajouter.

Commencez par y faire figurer &Fichier. Le "&" dit à Delphi de souligner le caractère qui suit et de définir la touche de raccourci correspondante. Ainsi, dans cet exemple vous pouvez appeler l'élément de menu Fichier en appuyant sur Alt et F pendant l'exécution. Une fois que vous avez ajouté l'élément de menu Fichier, vous pouvez remarquer qu'une boîte est apparue à la droite de cet élément. Elle vous permet en cliquant dessus d'ajouter un autre élément de menu pour peu que vous modifiiez à nouveau la propriété `Caption` de l'Inspecteur d'objets. Cliquez donc sur fichier. Cliquez dans la boîte qui apparaît alors et l'Inspecteur d'objets vous laisse modifier le titre. Ajoutez une option Quitter au menu Fichier. Modifiez la propriété `Caption` en `&Quitter` et Quitter s'ajoute en dessous de l'élément `Fichier`.

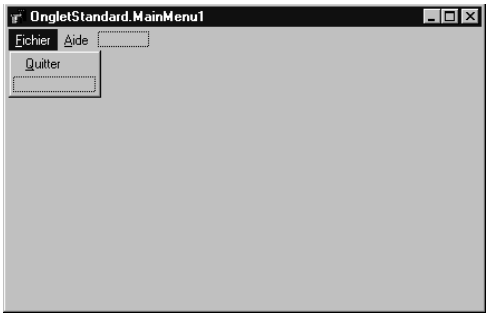
Remarquez les boîtes vides qui figurent sous Quitter et à droite de Fichier dans la boîte de menu. Vous cliquez sur celles-ci pour ajouter des options. Si vous ne cliquez pas dessus, elles n'apparaîtront pas dans votre menu.

Figure 8.10
L'Inspecteur d'objets pour le concepteur de menu.



Ajoutez un menu Aide comportant un élément Aide, une ligne séparatrice et un élément A propos. Pour ajouter le menu Aide, il suffit d'entrer &Aide dans les propriétés caption comme vous l'avez fait pour les éléments de menu Fichier et Quitter. Pour créer la ligne séparatrice, ajoutez un tirer (-) dans la propriété caption du sous-menu, juste sous l'élément Aide en l'occurrence. Vous verrez le séparateur apparaître immédiatement. Vous pouvez maintenant ajouter un élément A propos en ajoutant `&A propos` sous la ligne séparatrice (voir Figure 8.11).

Figure 8.11
Le Concepteur de menu comportant les éléments de menu.



Lorsque vous avez fini la conception du menu, cliquez sur la case de fermeture de l'écran Concepteur de menu. Le Concepteur de menu se ferme et vous pouvez voir vos éléments de menu sur la fiche. Vous pouvez alors naviguer dans ces éléments comme vous le feriez dans un menu normal en cliquant sur une option, en laissant le bouton de la souris enfoncé et en faisant glisser votre souris. Pour ajouter du code à un des éléments, il vous suffit de sélectionner l'élément concerner comme si vous étiez dans un menu, et vous vous retrouvez dans le Gestionnaire d'événements correspondant. Vous pouvez alors faire un Sélectionner Fichier/Quitter sur votre nouvel élément de menu et ajouter la section de code qui suit pour le refermer. Utilisez plutôt le Gestionnaire d'événements que vous avez créé auparavant. Dans l'Inspecteur d'objets, sélectionnez `Quitter1` (votre élément de menu Fichier/Quitter) en utilisant la boîte à options. Puis sélectionnez l'onglet Evénements dans l'Inspecteur d'objets et cliquez sur la flèche vers

le bas de la boîte à options de l'événement `OnClick` pour voir les options disponibles. Sélectionnez `Button3Click`. Ainsi, vous avez évité de retaper une ligne de code. Dans un Gestionnaire d'événements de taille plus importante, le temps gagné aurait été bien plus appréciable.

Ce gestionnaire se contente de fermer l'application. Vous n'avez pas besoin d'ajouter du code dans les autres éléments. Il ne s'agit que d'un exemple vous montrant comment créer des menus. Plus tard, nous reviendrons sur ces éléments et nous ajouterons le code correspondant aux éléments Aide et A propos de.

Note

Si vous n'avez pas laissé assez de place au sommet de votre fiche, il se peut que votre menu recouvre certains de vos composants et que les barres de défilement aient disparu. Pour y remédier, il suffit dans ce cas de redimensionner la fiche, en l'allongeant et en déplaçant les composants vers le bas pour laisser un peu de place au menu.

Il vous suffit d'ajouter le menu contextuel et vous en aurez fini avec cette fiche. Cliquez sur le composant `PopupMenu` que vous avez placé sur la fiche tout à l'heure. Cliquez à droite pour faire apparaître le menu contextuel. Là, sélectionnez Concepteur de menu... Ce concepteur de menu fonctionne de la même manière que celui du `MainMenu`, à la différence que tous les éléments de menu sont disposés verticalement le long d'une boîte unique. Pour ajouter du code aux éléments de menu du menu contextuel, vous devez sélectionner l'élément de menu dans le Concepteur. Vous vous retrouvez alors dans le code du Gestionnaire d'événements. Ajoutez les deux éléments Aller à et Mise à jour polices. Ceci fait, sortez du Concepteur. Utilisez pour ces éléments les méthodes déjà existantes, en reprenant la technique mise ne œuvre pour le menu précédent. Dans l'Inspecteur d'objets, sélectionnez Allez à et sélectionnez `Button1Click` pour l'événement `OnClick`. Ensuite, utilisez l'Inspecteur pour sélectionner `UpdateFonts1` et sélectionnez `Button2Click` pour l'événement `OnClick`. Votre menu fonctionne et vous n'avez pas eu à retaper le code.

Si tout s'est bien passé, vous devez avoir terminé avec cette partie de l'application VCLDEMO. Veillez à bien enregistrer votre travail. Vous pouvez maintenant regarder le code complexe de cette fiche, qui figure dans le Listing 8.1.

Listing 8.1 : Code de l'onglet standard de la fiche VCLDEMO

```
unit Standard;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Menus, StdCtrls, ExtCtrls,Addition;

type
  TStandardTab = class(TForm)
    MainMenu1: TMainMenu;
```

```
    PopupMenu1: TPopupMenu;
    Panel1: TPanel;
    Label1: TLabel;
    Edit1: TEdit;
    RadioGroup1: TRadioGroup;
    GroupBox1: TGroupBox;
    Button3: TButton;
    Button2: TButton;
    CheckBox1: TCheckBox;
    CheckBox2: TCheckBox;
    ScrollBar1: TScrollBar;
    Memo1: TMemo;
    ComboBox1: TComboBox;
    ListBox1: TListBox;
    Button1: TButton;
    Button4: TButton;
    RadioButton1: TRadioButton;
    RadioButton2: TRadioButton;
    File1: TMenuItem;
    Exit1: TMenuItem;
    Help1: TMenuItem;
    Help2: TMenuItem;
    N1: TMenuItem;
    About1: TMenuItem;
    Go: TMenuItem;
    UpdateFonts1: TMenuItem;
    procedure RadioGroup1Click(Sender: TObject);
    procedure ScrollBar1Change(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure GoClick(Sender: TObject);
    procedure UpdateFonts1Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
```

```
  StandardTab: TStandardTab;

implementation

{$R *.DFM}

procedure TStandardTab.RadioGroup1Click(Sender: TObject);
begin
     If RadioGroup1.ItemIndex=0 Then StandardTab.Color:=clSilver;
     If RadioGroup1.ItemIndex=1 Then StandardTab.Color:=clBlue;
     If RadioGroup1.ItemIndex=2 Then StandardTab.Color:=clYellow;
     If RadioGroup1.ItemIndex=3 Then StandardTab.Color:=clRed;
end;

procedure TStandardTab.ScrollBar1Change(Sender: TObject);
begin
     RadioGroup1.ItemIndex := ScrollBar1.Position;
end;

procedure TStandardTab.Button3Click(Sender: TObject);
begin
     Close;
end;

procedure TStandardTab.Button2Click(Sender: TObject);
begin
     If CheckBox1.State = cbChecked then
        StandardTab.Font.Style:=[fsItalic]
     else StandardTab.Font.Style:=[];
     If CheckBox2.State = cbChecked then
        StandardTab.Font.Color:=clPurple
     else StandardTab.Font.Color:=clBlack;
end;

procedure TStandardTab.Button1Click(Sender: TObject);
Var
   x : Integer;
begin
     Memo1.Clear;
     Memo1.Lines.Add(Edit1.Text);
     Memo1.Lines.Add(ComboBox1.Text);
```

```
     Memo1.Lines.Add(EListBox String#i+IntToStr(ListBox1.ItemIndex+1));
     If RadioButton1.Checked then Memo1.Color:=clWhite;
     If RadioButton2.Checked then Memo1.Color:=ClAqua;
end;

procedure TStandardTab.GoClick(Sender: TObject);
begin
     Button1Click(StandardTab);
end;

procedure TStandardTab.UpdateFonts1Click(Sender: TObject);
begin
     Button2Click(StandardTab);
end;

procedure TStandardTab.Button4Click(Sender: TObject);
begin
     StandardTab.Hide;
     AdditionalTab.Show;
end;
end.
```

Enregistrez votre projet et testez-le. Lorsque vous exécutez votre projet, la fiche doit ressembler à la Figure 8.12. Le seul élément de menu qui fasse quelque chose pour l'instant est Fichier/Quitter. Lorsque vous effectuez des modifications dans les composants `Edit`, `ListBox` ou `ComboBox` et appuyez sur le bouton Action, les mises à jour apparaissent dans le composant Memo. Lorsque vous sélectionnez Action, la couleur de fond devient celle sélectionnée par les boutons radio au bas de la fiche. Vous pouvez modifier la couleur du fond de la fiche en utilisant la barre de défilement ou en sélectionnant la couleur désirée parmi celles proposées par les boutons radio du `RadioGroup`. La couleur de police de la fiche peut devenir violette et la police passer en *italique* en cochant les cases à cocher et en cliquant sur le bouton Mise à jour police. Enfin, vous pouvez sélectionner Action ou Mise à jour polices dans le menu contextuel en cliquant à droite n'importe où dans la fiche.

Nous venons de voir tous les composants de l'onglet Standard et la façon dont chacun fonctionne. Vous pouvez remarquer que vous n'avez en fait pas écrit beaucoup de code et qu'il ne vous a pas été nécessaire de modifier de nombreuses propriétés pour que tout puisse fonctionner. Vous commencez peut-être à prendre la mesure de la puissance de la VCL de Delphi.

Figure 8.12
La fiche Onglet standard de VCLDEMO.

Onglet Supplément

L'onglet Supplément contient d'autres composants visuels fréquemment utilisés. Vous disposez dans cet onglet de treize composants. Nous allons les examiner et en utiliser certains dans notre projet VCLDEMO (voir Figure 8.13).

Figure 8.13
L'onglet Supplément.

Tableau 8.2 : L'onglet Supplément de la palette de composants

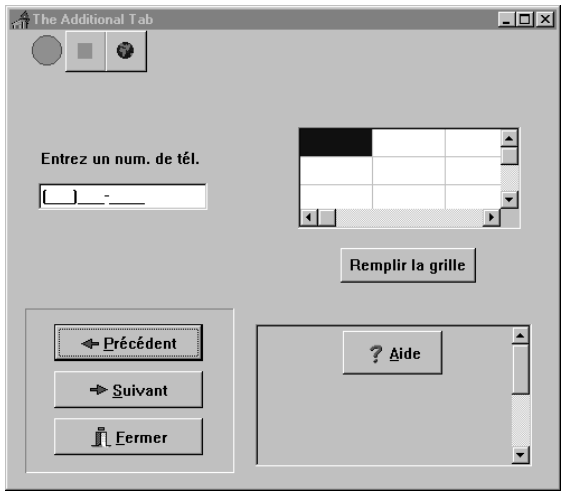
Composant	Description
	Le composant `BitBtn` permet de créer un bouton contenant un graphique bitmap (un bouton OK comportant une coche, par exemple).
	Le composant `SpeedButton` est conçu spécifiquement pour le composant Panel. Il vous permet de créer des barres d'outils et des ensembles dédiés de boutons, tels que des boutons qui restent enfoncés.

Composant	Description
	Le composant `MaskEdit` permet de formater des données ou sert de crible pour certaines entrées.
	Le composant `StringGrid` est utilisé pour afficher des données de chaînes en lignes et en colonnes.
	Le composant `DrawGrid` permet d'afficher des informations — autres que textuelles — dans des lignes et des colonnes.
	Le composant `Image` permet d'afficher des graphiques tels qu'icônes, bitmap et métafichiers.
	Le composant `Shape` permet de dessiner des formes telles que des carrés, des cercles, etc.
	Le composant `Bevel` permet de dessiner un rectangle apparaissant enfoncé ou surélevé.
	Le composant `ScrollBox` permet de créer une zone d'affichage que l'on peut faire défiler.
	Le composant `CheckListBox` combine les fonctionnalités d'un `ListBox` et d'un `CheckBox` en un seul composant.
	Le composant `Splitter` permet de créer des zones d'affichage redimensionnables par l'utilisateur.
	Le composant `StaticText` ressemble au composant `Label`, mais comporte des fonctionnalités supplémentaires, comme le style de bordure.
	Le composant `Chart` fait partie de la famille `TChart` et est utilisé pour créer des graphes.

Maintenant que vous avez brièvement vu les composants de l'onglet Supplément, nous allons construire la deuxième page de l'application VCLDEMO pour apprendre à les utiliser. Ajoutez une nouvelle fiche au projet VCLDEMO. Modifiez la propriété `Name` de la fiche pour qu'elle devienne ongletsupplement. Modifiez caption en "Onglet Supplément". Sélectionnez Fichier/Enregistrer sous et enregistrez la nouvelle fiche sous `SUPPL.PAS`. Au fur et à mesure que vous

ajouterez les composants, reportez-vous Figure 8.14 pour connaître leur emplacement et leur taille approximative.

Figure 8.14
La fiche Onglet supplément de VCLDEMO.



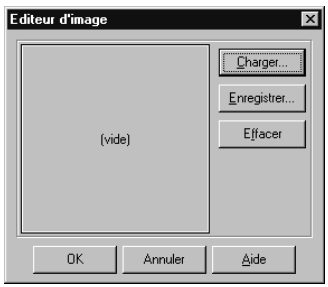
Centrez cette fiche sur l'écran. Pour ce faire, vous devez définir la propriété position. Cette propriété est définie par défaut comme `poDesigned`, qui signifie qu'elle apparaît dans la position de sa conception. Cliquez une fois sur `poDesigned` pour faire apparaître une boîte à options. Sélectionnez `poScreenCenter`. Ce paramètre prend effet lorsque la fiche est activée. Vous devez également ajouter Supplément à la clause `uses` des fiches. Avant de poursuivre, vous devez par ailleurs revenir à la fiche Onglet Standard pour ajouter du code vous permettant d'accéder à cette nouvelle fiche. Dans le menu, sélectionnez Voir/Fiches pour faire apparaître le menu Fiches, puis `OngletStandard`. Double-cliquez sur `Button4` (Page suivante) et définissez son événement `Click` comme suit :

```
procedure TStandardPage.Button4Click(Sender: TObject);
begin
     StandardPage.Hide;
     AdditionalPage.Show;
end;
```

Commençons par le composant `BitBtn`. Placez trois de ces composants dans le coin inférieur droit de la fiche. Pour `BitBtn1` et `BitBtn2`, définissez la propriété `Kind` à `bkCustom`. Définissez la propriété `caption` de `BitBtn1` comme `Préc.` et celle de `BitBtn2` comme `Suivant`. Le composant `BitBtn` vous permet de placer sur un bouton un graphique, appelé *glyphe*. Les glyphes ne sont rien d'autres que de petits bitmap pouvant contenir des images différentes, selon l'état d'un `BitBtn` ou d'un `SpeedButton`. Vous pouvez utiliser un des glyphes prédéfinis proposés

par Delphi ou concevoir le vôtre en utilisant l'Editeur d'image qui se trouve dans le menu Outils. Nous allons nous contenter des glyphes fournis par Delphi. Pour `BitBtn1`, double-cliquez sur `Tbitmap` dans la propriété `Glyph`. Vous faites ainsi apparaître l'Editeur d'image (voir Figure 8.15). Cet écran ne vous permet en fait pas d'éditer l'image, mais vous pouvez sélectionner le nom de fichier du glyphe désiré pour le visualiser avant de l'ajouter au bouton.

Figure 8.15
L'Editeur d'image.



Cliquez sur le bouton Charger pour faire apparaître la boîte de dialogue de sélection de fichier. Allez dans le répertoire C:\program files\borland\delphi 2.0\images\buttons (si votre configuration est standard) et sélectionnez ARROW1.BMP. Vous voyez apparaître une flèche rouge pointant vers la gauche ainsi qu'une flèche blanche. La flèche rouge s'affiche lorsque le bouton est dans l'état `enabled` (activé). La flèche blanche quand le bouton est en état `disabled` (désactivé). Cette flèche donne l'aspect grisé au bouton. Vous pouvez faire basculer la propriété `Enabled` du bouton entre `True` et `False` pour voir la différence d'aspect qui en résulte. Lorsque vous avez terminé, assurez-vous que la propriété `Enabled` du bouton est définie comme `True`. En suivant la même méthode pour `BitBtn2`, définissez son glyphe comme ARROW1R.BMP. La définition de `BitBtn3` sera plus facile car vous utilisez un type prédéfini dans la propriété `Kind`. Pour `BitBtn3`, cliquez sur `Kind` et sélectionnez `bkClose`. Ceci définit le glyphe du bouton comme une porte ouverte et la propriété `Caption` comme Fermer. La propriété `Kind` comporte 10 types prédéfinis auxquels s'ajoute un type `bkCustom` qui vous permet de créer votre propre type de bouton.

- `bkAbout`
- `bkAll`
- `bkCancel`
- `bkClose`
- `bkCustom`
- `bkHelp`
- `bkIgnore`
- `bkNo`
- `bkOk`
- `bkRetry`
- `bkYes`

Les noms sont relativement parlants. Ces types correspondent à des boutons usuels de Windows. Le type `bkCustom` est utilisé lorsque vous souhaitez créer un nouveau type de bouton. Regroupez correctement vos boutons en ajoutant un composant `Bevel`. Sélectionnez le composant `Bevel` dans l'onglet supplément et tracez-le autour de la boîte. Vous donnez ainsi l'impression que votre boîte est légèrement enfoncée.

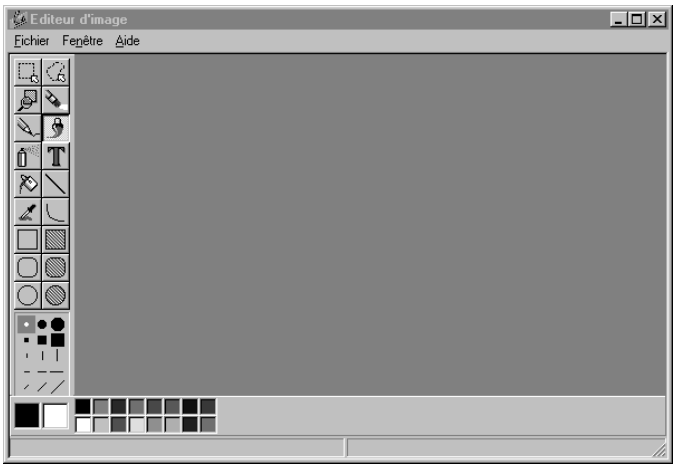
Ajoutez un peu de code aux boutons en double-cliquant sur `BitBtn1` et en ajoutant le code suivant à l'événement `Click` :

```
procedure TAdditionalTab.BitBtn1Click(Sender: TObject);
begin
   Standard.StandardTab.Show;
   AdditionalTab.Hide;
end;
Nous reviendrons sur BitBtn2 plus tard, mais pour l'instant, ajoutez ce code
à bitBtn3 :
procedure TAdditionalTab.BitBtn3Click(Sender: TObject);
begin
  Standard.StandardTab.Close;
end;
Pour que le bouton Fermer fonctionne, ajoutez le code suivant à l'événement
Close de la fiche :
procedure TAdditionalTab.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
     Application.Terminate;
end;
```

Si vous n'ajoutez pas ce code, la fiche se refermera lorsqu'on appuiera sur le bouton Fermer, mais l'application restera en mémoire, mobilisant inutilement des ressources. A présent, créez une barre d'outils. Pour ce faire, vous devez utiliser un composant `Panel` comme conteneur de vos boutons. Ajoutez un `Panel` près du coin supérieur gauche de la fiche, en vous assurant qu'il est assez grand pour contenir deux boutons carrés. Ajoutez ensuite deux `SpeedButton` au volet (Panel). Les `SpeedButtons` vous permettent de faire certaines choses impossibles avec d'autres boutons. Ainsi, vous pouvez créer des boutons qui restent enfoncés, vous pouvez regrouper des boutons et même créer des boutons dont le glyphe change selon l'état du bouton. Les états disponibles pour un `SpeedButton` sont `Up` (relevé), `Disabled` (désactivé), `Down` (enfoncé) et `Stay Down` (toujours enfoncé). A chacun de ces états correspond un glyphe. Pour tirer partie de ces états, vous pouvez utiliser l'Editeur d'image pour créer un glyphe composé de quatre images. Nous avons vu très brièvement l'Editeur d'image au 2ème jour (reportez-vous aux manuels ou à l'aide en ligne pour plus de détails). L'Editeur d'image est un programme de dessin simple, conçu pour créer des ressources, des icônes, des bitmap, etc. (voir Figure 8.16). Si vous avez déjà utilisé un programme Windows tel que Paint ou Draw, vous ne

serez pas dépaysé par l'Editeur d'image. Si vous ne vous sentez pas prêt à utiliser cet éditeur tout de suite, vous pouvez poursuivre la conception et revenir à l'éditeur un peu plus tard.

Figure 8.16
L'Editeur d'image.



Vous allez créer un bitmap 16 couleurs de taille 16 × 64. L'image comportera quatre boîtes carrées de même taille mais de couleurs différentes. En allant de gauche à droite, vous pouvez les colorer en vert, gris, violet et rouge respectivement. Lorsque vous avez terminé, il vous suffit de faire Fichier/Enregistrer, de définir le type comme BMP et d'enregistrer le glyphe sous un nom de votre choix par exemple GRNRED.BMP par exemple. En utilisant la même méthode que pour le BitBtn, définissez la propriété Glyph de SpeedButton1 comme GRNRED.BMP, qui est le fichier que vous venez de créer. Définissez la propriété Glyph de SpeedButton2 comme C:\program files\borland\delphi 2.0\images\buttons\globe.bmp (si votre configuration d'installation est standard). Pour le SpeedButton1, assurez-vous que la propriété NumlGlyphs est définie à 4 (4 est la valeur par défaut). Ceci indique qu'il y a quatre glyphes disponibles pour les 4 états mentionnés plus haut. Définissez la propriété `GroupIndex` à 1. Vous indiquez ainsi au bouton qu'il appartient au groupe numéro 1. Un seul bouton peut être enfoncé à la fois dans un groupe. Comme dans le cas des boutons radio, si vous cliquez sur l'un des boutons, tous les autres boutons du groupe se relèvent. Définissez à 2 les propriétés `GroupIndex` et `NumGlyphs` du `SpeedButton2`. Vous venez de finir la conception de votre barre d'outils, il ne reste plus qu'à ajouter le code.

Double-cliquez sur `SpeedButton1` et faites que son événement `Click` ressemble à :

```
procedure TAdditionalTab.SpeedButton1Click(Sender: TObject);
begin
     If SpeedButton1.Down=True then
```

```
        Begin
           Image1.Visible:=False;
           Shape1.Brush.Color:=clRed;
        end;
end;

L'événement Click du SpeedButton2 sera celui-ci :
procedure TAdditionalTab.SpeedButton2Click(Sender: TObject);
begin
     If SpeedButton2.Down=True then
        Begin
           Image1.Visible:=True;
           Shape1.Brush.Color:=clLime;
        end;
end;
```

Ajoutez maintenant un composant `Shape` à gauche de la barre d'outils. Donnez-lui à peu près la taille des SpeedButtons de la barre d'outils. Définissez sa propriété `Shape` comme `stEllipse`. Double-cliquez sur la propriété `Pen` pour faire apparaître les propriétés imbriquées. Définissez la propriété `Color` comme `clGreen`. Double-cliquez sur la propriété `Brush` pour faire apparaître ses propriétés imbriquées et définissez sa propriété `Color` comme `clLime`. Les propriétés `Pen` définissent les attributs utilisés lors du dessin de la forme (couleur de la ligne, etc). Les propriétés `Brush` définissent les attributs utilisés pour remplir la forme.

Ensuite, ajoutez un composant Image à droite de la barre d'outils. Faites que ce composant image ait à peu près la taille des SpeedButtons. Le composant `Image` peut afficher des graphiques tels que des icônes, des bitmap, ou des métafichiers. Ici, nous utiliserons une icône. Double-cliquez sur la propriété Picture pour faire apparaître l'Editeur d'image. Chargez un fichier d'icône dans \delphi\images\icons\earth. Le code que vous avez ajouté aux SpeedButtons permet d'affecter l'aspect des composants `Shape` et `Image`.

Sur le `SpeedButton1`, utilisez `Hint`, une propriété commune à tous les composants visuels. Cette propriété `Hint` stocke une chaîne de texte qui s'affiche lorsque l'utilisateur laisse le pointeur de la souris au-dessus du composant pendant un certain temps. Modifiez la propriété `Hint` du bouton en `Infobulle`. Pour activer la propriété `Hint`, vous devez définir comme `True` la propriété `ShowHints` (qui est `False` par défaut). Vous pourrez tester l'info-bulle lors de l'exécution.

Ajoutez un composant Label à la fiche, avec pour propriété Caption le texte Entrez un numéro de téléphone. Ensuite, juste au-dessous du label, ajoutez un composant MaskEdit. Double-cliquez sur la propriété `EditMask` pour faire apparaître l'Editeur de masque de saisie. Cliquez sur `Phone` et appuyez sur OK. De cette manière, le composant `EditMask` n'acceptera que des chiffres dans un format correspondant à un indicatif suivi d'un numéro de téléphone.

Placez un composant `StringGrid` sur la fiche. Définissez à 3 les propriétés `RowCount` et `ColCount`, et à 0 les propriétés `FixedCols` et `FixedRows`. Ajustez la taille de la `StringGrid` pour

que seules neuf cellules soient visibles. Ajoutez un bouton à la fiche et donnez-lui pour `caption` `Remplir la grille`. Double-cliquez sur le bouton et ajoutez le code ci-après :

```
procedure TAdditionalTab.Button1Click(Sender: TObject);
var
  x, y: Integer;
begin
  with StringGrid1 do
    for x := 0 to ColCount - 1 do
      for y:= 0 to RowCount - 1 do
        Cells[x,y] := 'Cord. '+ IntToStr(x)+i-i+IntToStr(y);
end;
```

Le dernier composant de cette page est la `ScrollBox`. Tracez cette `ScrollBox` dans le coin inférieur droit de la fiche en vous référant à la Figure 8.14 pour sa taille. Vous allez ajouter quelques composants à cette zone afin de montrer que l'on peut la faire défiler. Ajoutez un `BitBtn` à la boîte et définissez sa propriété `Kind` comme `bkHelp`. Ajoutez un `Panel` à la boîte. Si vous faites glisser le Panel sous la bordure inférieure de la boîte de défilement, des barres de défilement apparaissent. Etirez temporairement la boîte de défilement de bas en haut, jusqu'à lui faire prendre une taille deux ou trois fois plus grande que nécessaire. Placez le bouton au sommet et le volet au bas. Définissez la propriété `Caption` du volet comme `Le volet de la boîte de défilement`. Ensuite, redimensionnez la boîte de défilement pour qu'elle ressemble à celle de la Figure 8.13. Double-cliquez sur le bouton d'aide de la boîte de défilement et ajoutez le code ci-après à son événement `Click` :

```
procedure TAdditionalTab.BitBtn4Click(Sender: TObject);
begin
     ShowMessage('Test du bouton Aide dans la Scrollbox');
end;
Double-cliquez sur BitBtn2 et ajoutez le code ci-après :
procedure TAdditionalTab.BitBtn2Click(Sender: TObject);
begin
     ShowMessage('Cette fonction n''est pas active');
end;
```

Vous en avez fini avec la fiche Onglet supplément et vous pouvez la tester. Le code source complet de l'unité Supplément que vous avez créée doit ressembler au Listing 8.2 :

Listing 8.2 : Code de l'onglet Supplément de VCLDEMO

```
unit Addition;

interface

```

```
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Buttons, StdCtrls, ExtCtrls, Grids, Outline, Mask,
  TabNotBk, Tabs;

type
  TAdditionalTab = class(TForm)
    BitBtn1: TBitBtn;
    BitBtn2: TBitBtn;
    Panel1: TPanel;
    SpeedButton1: TSpeedButton;
    SpeedButton2: TSpeedButton;
    Bevel1: TBevel;
    Image1: TImage;
    Shape1: TShape;
    ScrollBox1: TScrollBox;
    BitBtn3: TBitBtn;
    BitBtn4: TBitBtn;
    Panel5: TPanel;
    Label1: TLabel;
    MaskEdit1: TMaskEdit;
    StringGrid1: TStringGrid;
    Button1: TButton;
    procedure BitBtn1Click(Sender: TObject);
    procedure BitBtn3Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure SpeedButton2Click(Sender: TObject);
    procedure SpeedButton1Click(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure BitBtn4Click(Sender: TObject);
    procedure BitBtn2Click(Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  AdditionalTab: TAdditionalTab;

implementation
```

```
Uses
     Standard;

{$R *.DFM}

procedure TAdditionalTab.BitBtn1Click(Sender: TObject);
begin
   StandardTab.Show;
   AdditionalTab.Hide;
end;

procedure TAdditionalTab.BitBtn3Click(Sender: TObject);
begin
  StandardTab.Close;
end;

procedure TAdditionalTab.Button1Click(Sender: TObject);
var
  x, y: Integer;
begin
  with StringGrid1 do
    for x := 0 to ColCount - 1 do
      for y:= 0 to RowCount - 1 do
        Cells[x,y] := 'Cord. '+ IntToStr(x)+i-i+IntToStr(y);
end;

procedure TAdditionalTab.SpeedButton2Click(Sender: TObject);
begin
     If SpeedButton2.Down=True then
        Begin
           Image1.Visible:=True;
           Shape1.Brush.Color:=clLime;
        end;
end;

procedure TAdditionalTab.SpeedButton1Click(Sender: TObject);
begin
     If SpeedButton1.Down=True then
        Begin
           Image1.Visible:=False;
```

```
            Shape1.Brush.Color:=clRed;
        end;
end;

procedure TAdditionalTab.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
     Application.Terminate;
end;

procedure TAdditionalTab.BitBtn4Click(Sender: TObject);
begin
     ShowMessage('Test du bouton Aide!');
end;

procedure TAdditionalTab.BitBtn2Click(Sender: TObject);
begin
     ShowMessage('Cette fonctionnalité n''est pas implémentée');
end;

end.
```

Enregistrez le projet et testez-le. Vous devriez pouvoir parcourir les deux fiches et tester les composants que vous avez placés sur chacune des pages. Cliquez sur les boutons de la barre d'outils (les SpeedButtons) et observez les divers résultats. Utilisez la barre de défilement pour voir le bouton Aide et le volet. Cliquez sur le bouton Aide pour faire apparaître une boîte de message. Lorsque vous avez terminé, cliquez sur Fermer et vous quittez le programme.

Vous avez pu voir de nombreux composants et vous avez beaucoup travaillé, sans pour autant avoir construit quelque chose de véritablement utile. Le but était de vous familiariser avec la VCL et de vous donner un peu de pratique. Vous attendez peut-être avec appréhension les exercices suivants consacrés aux sept autres onglets... Rassurez-vous, le plus dur est derrière vous. Les composants `AccèsDB` et `ContrôleDB` ne seront abordés que brièvement ici (ils seront détaillés dans la partie consacrée aux bases de données). Nous allons parler de l'onglet Win95, de l'onglet Dialogues et de l'onglet Système. D'autres composants de cet onglet Système seront détaillés dans d'autres journées. Nous ne parlerons pas des composants des onglets ActiveX et Exemple, car il ne s'agit là que de composants exemple, et ils ne sont pas pris en charge ou documentés officiellement. Examinons rapidement le contenu des autres onglets et construisons une petite application qui permettra de tester certains composants qui s'y trouvent.

Onglet Win32

L'onglet WIN32 contient 16 composants qui vous permettent de créer des applications dans l'esthétique de Windows 95 et NT 4.0 (voir Figure 8.17). Certains de ces contrôles sont similaires à ceux que l'on peut trouver dans l'onglet Win3.1. Tous, à l'exception de ImageList, sont des composants visuels. Le Tableau 8.3 décrit brièvement chaque composant.

Figure 8.17
L'onglet Win32.

Tableau 8.3 : Les composants de l'onglet Win32

Composant	Description
	`TabControl` est un composant de tabulation Windows95 permettant d'ajouter des tabulations à une fiche pour que l'utilisateur les sélectionne.
	`PageControl` est un composant Windows 95 qui permet de créer des pages qui peuvent être modifiées à l'aide de tabulations ou d'autres contrôles tout en préservant l'espace du bureau.
	`TreeView` est un composant Windows95 qui permet d'afficher des données dans un format hiérarchique.
	`ListView` est un composant Windows95 qui affiche des listes en colonnes.
	`ImageList` est un nouvel objet qui permet de travailler avec des listes d'images. Pour plus de détails, reportez-vous à la documentation de Delphi.
	`Header` est un composant Windows 95 permettant la création de plusieurs en-têtes déplaçables.
	`RichEdit` est une boîte d'édition Windows95 qui permet d'utiliser plusieurs couleurs et polices, d'effectuer des recherches sur le texte, etc.

Composant	Description
	`StatusBar` est une barre d'état Windows95 permettant d'afficher des informations d'état dans plusieurs volets si nécessaire.
	`TrackBar` est un composant de barre de curseur de type Windows 95.
	`ProgressBar` est un composant de barre de progression de type Windows 95.
	`UpDown` est un composant de bouton de réglage de type Windows 95.
	`HotKey` permet l'ajout des touches de raccourcis dans une application.
	`Animate` est un composant vous permettant de jouer des fichiers AVI silencieux.
	`DatePicker` est un composant vous permettant de choisir graphiquement une date.
	`ToolBar` est un composant permettant de créer des barres d'outils.

Onglet Système

L'onglet Système contient 8 composants permettant de tirer parti des fonctionnalités même de Windows. Les composants `PaintBox` et `MediaPlayer` seront traités dans d'autres parties de cet ouvrage. Les composants `DDE` et `OLE` sortent du cadre de cet ouvrage. Des exemples d'utilisation de ces composants figurent dans l'application qui clôt cette journée.

Figure 8.18
L'onglet Système.

Tableau 8.4 : Les composants de l'onglet Système

Composant	Description
	`Timer` permet d'activer des procédures, des fonctions et des événements à des intervalles de temps spécifiés. C'est un composant non visuel.
	`PaintBox` permet de créer dans la fiche une zone où l'on peut dessiner. C'est un composant visuel.
	`MediaPlayer` permet de créer un tableau de bord semblable à celui d'un magnétoscope. Ce contrôle permet de lire des fichiers son et vidéo. C'est un composant visuel.
	`OLEContainer` permet de créer une zone client OLE. C'est un composant visuel.
	`DDEClientConv` permet à un client DDE d'entamer une conversation avec un serveur DDE. C'est un composant non visuel.
	`DDEClientItem` permet de spécifier les données du client qui seront envoyées au serveur DDE lors d'une conversation. C'est un composant non visuel.
	`DDEServerConv` permet à une application serveur DDE d'entamer une conversation avec un client DDE. C'est un composant non visuel.
	`DDEServerItem` permet de spécifier des données qui seront envoyées à un client DDE lors d'une conversation. C'est un composant non visuel.

Onglet Internet

L'onglet Internet, visible Figure 8.19, comporte 15 composants qui facilitent grandement l'écriture de programmes Internet et TCP/IP. Le Tableau 8.5 décrit brièvement chaque composant.

Figure 8.19
L'onglet Internet.

Tableau 8.5 : Les composants de l'onglet Internet

Composant	Description
	`FTP` est utilisé pour transférer des fichiers entre ordinateurs via l'Internet ou un autre réseau TCP/IP, et ce grâce au protocole FTP.
	`HTML` est utilisé pour afficher des pages HTML, comme le fait un logiciel de navigation Web.
	`HTTP` est utilisé pour se connecter à des serveurs Web.
	`POP` est utilisé pour se connecter à des serveurs POP afin de récupérer du courrier électronique.
	`TCP` est utilisé pour communiquer avec d'autres ordinateurs sur un même réseau TCP/IP.
	`UDP` est utilisé pour effectuer des connexions UDP sur un réseau TCP/IP.
	`ClientSocket` est utilisé pour établir une connexion avec une autre machine à travers un réseau.
	`ServerSocket` est utilisé pour répondre à des requêtes en provenance d'autres machines sur un réseau.
	`WebDispatcher` est utilisé pour convertir un module standard de données en module Web.

Composant	Description
	`PageProducer` est utilisé pour convertir un modèle HTML en une chaîne de code HTML qui peut être visualisée sur un logiciel de navigation Web ou tout autre visualisateur HTML.
	`QueryTableProducer` est utilisé pour créer un Tableau HTML à partir des enregistrements d'un objet `Tquery`.
	`DataSetTableProducer` est utilisé pour créer un Tableau HTML à partir des enregistrements d'un `TDataSet`.

Onglet AccèsBD

L'onglet AccèsBD visible Figure 8.20 contient des composants non visuels permettant de se relier et de communiquer à des bases de données. Ces composants ne seront pas décrits ici car ils feront l'objet d'une partie entière dans la suite de cet ouvrage (voir Tableau 8.6).

Figure 8.20
L'onglet AccèsBD.

Tableau 8.6 : Les composants de l'onglet AccèsBD

Composant	Description
	`DataSource` permet de connecter les composants `Table` ou `Query` à des composants de bases de données.
	`Table` permet de lier une base de données à une application.
	`Query` permet de créer et d'exécuter des requêtes SQL faites à un serveur SQL distant ou à une base de données locale.

Composant	Description
	`StoredProc` permet d'exécuter des procédures qui ont été stockées sur un serveur SQL.
	`Database` permet d'établir des connexions avec des serveurs de bases de données distants ou locaux.
	`Session` permet de contrôler globalement les connexions de base de données d'une application.
	`BatchMove` permet de manipuler localement des enregistrements et des tables, puis de rapatrier dans le serveur les informations mises à jour.
	`UpdateSQL` permet d'effectuer des mises à jour dans une base de données SQL.
	`Provider` est utilisé pour mettre à disposition une connexion entre un serveur distant de bases de données et un ensemble de données client dans une application de bases de données multiliaison.
	`ClientDataset` effectue l'accès client à une base de données multiliaison.
	`RemoteServer` est utilisé pour se connecter à une base de données multiliaison sur un serveur distant.

Onglet ContrôleBD

L'onglet ContrôleBD contient différents composants visuels orientés données. La plupart de ces composants sont des version orientées données de composants usuels qui figurent dans les onglets Standard et Supplément (voir Figure 8.21). Nous décrivons rapidement ces composants dans le Tableau 8.7.

Figure 8.21
L'onglet ContrôleBD.

Tableau 8.7 : Les composants de l'onglet ContrôleBD

Composant	Description
	`DBGrid` permet de créer une grille orientée données dans laquelle on peut afficher des données en lignes et en colonnes.
	`DBNavigator` permet de créer un contrôle de navigation doté de capacités d'édition dans la base de données.
	`DBText` est une version orientée données du composant `Label`.
	`DBEdit` est une version orientée données du composant `Edit`.
	`DBMemo` est une version orientée données du composant `Memo`.
	`DBImage` est une version orientée données du composant `Image`.
	`DBListBox` est une version orientée données du composant `ListBox`.
	`DBComboBox` est une version orientée données du composant `ComboBox`.
	`DBCheckBox` est une version orientée données du composant `CheckBox`.
	`DBRadioGroup` est une version orientée données du composant `RadioGroup`.
	`DBLookupListBox` permet de créer une `ListBox` de balayage (lookup ListBox) orientée données.
	`DBLookupComboBox` permet de créer une `ComboBox` de balayage (lookup ComboBox) orientée données.

Composant	Description
	`DBRichEdit` permet de créer un champ `RichEdit` orienté données.
	`DBLookupCtrlGrid` permet de créer une `CtrlGrid` de balayge (lookup CtrlGrid) orientée données.

Onglet Decision Cube

L'onglet Decision Cube, représenté Figure 8.22, comporte six composants de graphes destinés à simplifier l'écriture de logiciels d'analyse des données. Ces composants ne sont disponibles que dans la version CS de Delphi 3, et nous n'en donnerons ici qu'une description brève. Référez-vous à l'aide en ligne ou aux manuels fournis pour avoir plus de détails.

Figure 8.22
L'onglet Decision Cube.

Tableau 8.8 : Les composants de l'onglet Decision Cube

Composant	Description
	`DecisionCube` est un composant de stockage multidiensionnel de données qui peut être utilisé pour récupérer des données en provenance d'un ensemble de données.
	`DecisionQuery` est un composant spécialisé de `Tquery`, destiné à s'interfacer avec le `DecisionCube`.
	`DecisionSource` est un composant utilisé pour définir le pivot des composants `DecisionGrid` et `DecisionGraph`.
	`DecisionPivot` est un composant utilisé pour ouvrir ou fermer des dimensions (ou champs) du composant `DecisionCube` en pressant sur un bouton.

Composant	Description
	`DecisionGrid` est un composant utilisé pour afficher dans une grille les données d'un composant `DecisionCube` lié à un ensemble de données `DecisionCube`.
	`DecisionGraph` est utilié pour afficher dans un graphe les données d'un composant `DecisionCube` lié à un ensemble de données `DecisionCube`.

Onglet QReport

L'onglet QReport contient 18 composants destinés à la génération de rapports, comme indiqué Figure 8.23. Le Tableau 8.9 en donne une présentation sommaire.

Figure 8.23
L'onglet QReport.

Tableau 8.9 : Les composants de l'onglet QReport

Composant	Description
	`QuickReport` permet d'ajouter des capacités d'impression `QuickReport` à des applications. C'est un composant non visuel.
	`QRSubDetail` permet de lier des ensembles de données supplémentaires à un état. C'est un composant visuel.
	`QRBand` permet de construire des rapports en y plaçant des composants imprimables. C'est un composant visuel.
	`QRChildBand` est utilisé pour créer des bandes enfant qui peuvent contenir d'autres composants QuickReport. C'est un composant visuel.
	`QRGroup` permet de créer des groupes de données. C'est un composant non visuel.

Composant	Description
	`QRLabel` permet de placer du texte dans un rapport. C'est un composant visuel.
	`QRDBText` est un composant orienté données permettant de placer du texte dans un rapport. C'est un composant visuel.
	`QRExpr`
	`QRSysData` permet d'afficher des données système. C'est un composant visuel.
	`QRMemo` permet de placer du texte (multiligne) dans un rapport. C'est un composant visuel.
	`QRRichText` est un composant utilisé pour insérer un champ `RichText` dans un un état. C'est un composant visuel.
	`QRDBRichText` est un composant utilisé pour insérer du texte en provenance d'un champ `RichText` d'une base de données et à destination d'un état. C'est un composant visuel.
	`QRShape` permet de dessiner une forme sur un rapport. C'est un composant visuel.
	`QRImage` est un composant utilisé pour afficher des images dans un état. C'est un composant visuel.
	`QRDBImage` est un composant utilisé pour afficher des images en provenance d'une base de données et à destination d'un état. C'est un composant visuel.
	`QRCompositeReport` est un composant utilisé pour créer des états composites. C'est un composant non visuel.
	`QRPreview` permet de créer des fiches d'aperçu pour voir à l'écran les rapports. C'est un composant visuel.

Composant	Description
	`QRChart` est utilisé pour afficher des diagrammes et des graphiques dans un état. C'est un composant visuel.

Onglet Dialogues

L'onglet Dialogues, repris Figure 8.24, contient dix composants non visuels permettant de créer les diverses boîtes de dialogue usuelles. Les boîtes de dialogues permettent de spécifier des fichiers ou de sélectionner des paramètres. En utilisant celles fournies par Delphi, vous gagnerez du temps et donnerez un aspect uniforme et cohérent à toutes vos applications. Pour plus de détails, cliquez sur un de ces composants et appuyez sur F1. Ces composants sont sommairement décrits Tableau 8.10.

Figure 8.24
L'onglet Dialogue.

Tableau 8.10 : Les composants de l'onglet Dialogues

Composant	Description
	`OpenDialog` permet de créer une boîte de dialogue Ouvrir fichier.
	`SaveDialog` permet de créer une boîte de dialogue Enregistrer fichier.
	`OpenPictureDialog` permet d'ouvrir une boîte de dialogue Ouvrir image.
	`SavePictureDialog` permet d'ouvrir une boîte de dialogue Enregistrer image.
	`FontDialog` permet de créer une boîte de dialogue Polices.

Composant	Description
	`ColorDialog` permet de créer une boîte de dialogue Couleurs.
	`PrintDialog` permet de créer une boîte de dialogue Imprimer.
	`PrinterSetupDialog` permet de créer une boîte de dialogue Configuration de l'impression.
	`FindDialog` permet de créer une boîte de dialogue Rechercher.
	`ReplaceDialog` permet de créer une boîte de dialogue Remplacer.

Onglet WIN3.1

L'onglet Win31 (voir Figure 8.25) contient des composants visuels maintenant obsolètes. Ces composants sont disponibles pour des raisons de compatibilité ascendante lors du portage d'applications de Delphi 1.0 et 2.0 vers Delphi 3.0. Ils ne doivent pas être utilisés dans des applications 32 bits. Le Tableau 8.11 décrit rapidement chacun de ces composants.

Figure 8.25
L'onglet Win31.

Tableau 8.11 : Les composants de l'onglet Win31

Composant	Description
	`DBLookupList` est un composant orienté données de Windows 3.1 qui permet de consulter une valeur dans une table à l'aide d'une boîte de liste.

Composant	Description
	`DBLookupCombo` est un composant orienté données de Windows 3.1 qui permet de consulter une valeur dans une table à l'aide d'un boîte à options.
	`TabSet` permet de créer des onglets de bloc-notes.
	`Outline` permet de créer un contrôle d'arborescence affichant des données sous forme hiérarchique.
	`TabbedNoteBook` permet de créer des fiches multipages comportant des onglets.
	`Notebook` permet de créer un ensemble de pages qui s'utilisent avec un composant `TabSet`.
	`Header` permet de créer un contrôle qui affiche un texte dans des parties redimensionnables.
	`FileListBox` est utilisé pour créer une boîte de liste destinée à afficher les fichiers dans le répertoire ou dans le lecteur sélectionné.
	`DirectoryListBox` est utilisé pour créer une boîte de liste destinée à afficher les répertoires dans le lecteur sélectionné.
	`DriveComboBox` est utilisé pour créer une boîte destinée à afficher les lecteurs disponibles.
	`FilterComboBox` permet de créer un filtre d'affichage des fichiers.

Onglet Exemples

L'onglet Exemples contient sept composants d'exemple (comme indiqué Figure 8.26), qui sont présentés à titre indicatif et sans documentation étendue. Leur code source se trouve dans le répertoire \delphi\source\samples. Le Tableau 8.12 les décrit brièvement.

Figure 8.26
L'onglet Exemples.

Tableau 8.12 : Les composants de l'onglet Exemples

Composant	Description
	`Gauge` permet de créer un indicateur de progression apparaissant sous forme de barre, de texte ou de camembert. C'est un composant visuel.
	`ColorGrid` permet de créer une grille de couleurs dans laquelle l'utilisateur peut effectuer des sélections. C'est un composant visuel.
	`SpinButton` permet de créer des boutons de réglage. C'est un composant visuel.
	`SpinEdit` permet de créer une boîte d'édition dotée des fonctions d'un contrôle de réglage. C'est un composant visuel.
	`DirectoryOutline` permet de créer un affichage de la structure de répertoire du lecteur sélectionné. C'est un composant visuel.
	`Calendar` permet d'afficher un calendrier. C'est un composant visuel.
	`IBEventAlerter` est un composant d'alerte. C'est un composant non visuel.

Onglet ActiveX

L'onglet ActiveX (Figure 8.27) contient cinq exemples de contrôles ActiveX (ces contrôles ne font pas partie de la VCL de Delphi). Par ailleurs, c'est dans cet onglet que sont stockés les ActiveX que vous ajoutez à Delphi. Il est préférable d'utiliser les composants visuels de Delphi, mais si vous avez vraiment besoin d'utiliser un contrôle ActiveX, n'hésitez pas. N'oubliez cependant pas que si vous utilisez un ActiveX dans votre application Delphi, vous devez l'inclure avec votre application lors de la distribution. Ces composants ne sont pas détaillés, puisqu'il ne s'agit en fait que d'exemples. Le Tableau 8.13 les passe en revue.

Figure 8.27
L'onglet ActiveX.

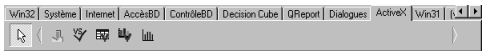


Tableau 8.13 : Les composants de l'onglet ActiveX

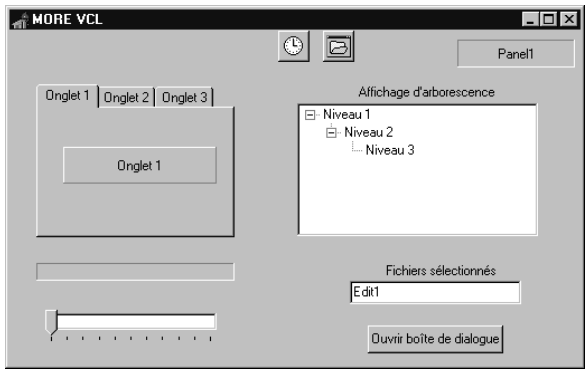
Composant	Description
	`ChartFX` permet d'ajouter des histogrammes dans une application Delphi. C'est un contrôle visuel.
	`VCFirstImpression` permet d'ajouter des fonctionnalités 3D à une application Delphi. C'est un contrôle visuel.
	`VCFormulaOne` permet d'ajouter des feuilles de calcul à une application Delphi. C'est un contrôle visuel.
	`VCSpeller` permet d'ajouter des fonctions de vérification orthographique à une application Delphi. C'est un contrôle non visuel.
	`GraphicsServer` permet lui aussi de générer des graphiques. C'est un contrôle visuel.

Dernier exemple

Maintenant que nous avons parcouru tous les onglets, nous allons construire une petite application utilisant quelques composants qui ne figuraient pas dans le projet VCLDEMO.DPR. Nous allons choisir nos composants dans les onglets Win32, Dialogues et Système. Dans l'onglet Win32, nous prendrons les composants `ProgressBar`, `TrackBar`, `TreeView` et `TabControl`. Dans l'onglet Dialogues, le composant `OpenDialog`. Enfin, nous utiliserons le composant `Timer` de l'onglet Système, ainsi que d'autres composants faisant partie de l'onglet Standard.

Notre nouveau projet s'appelle MOREVCL.DPR dont l'unité est MORE.PAS. Référez-vous à la Figure 8.28 pour construire votre fiche. Sur le côté gauche de la fiche, de haut en bas, placez les composants suivants : `TabControl`, `Panel` (que vous placez sur le `TabControl`), `ProgressBar` et `TrackBar`. Placez des composants `Timer` et `OpenDialog` sur la fiche (n'importe où, puisque ce sont des composants non visuels). Dans le coin supérieur droit, tracez un `Panel`. Puis, sur le côté droit, de haut en bas, placez un `Label`, un `TreeView`, un `Edit` et un `Button`.

Figure 8.28
La fiche MOREVCL.



Pour que cette fiche fonctionne, vous devez définir quelques propriétés et ajouter un peu de code. Pour le composant `TabControl`, double-cliquez sur la propriété `Tabs` de l'Inspecteur d'objets et faites apparaître la boîte d'édition `TStrings`. Là, vous pouvez entrer le noms des onglets. Chaque nom va sur une ligne différente. Entrez Onglet 1, Onglet 2 et Onglet 3. Pour le volet que vous avez placé dans le `TabControl`, donnez-lui comme caption Tab = 1. Ensuite, définissez à 10 les propriétés `Max` des composants `TrackBar` et `ProgressBar`. Ajoutez le code ci-après à l'événement `OnChange` de la `TrackBar` :

```
ProgressBar1.Position:=TrackBar1.Position;
```

Ensuite, double-cliquez sur le timer et ajoutez le code ci-après à l'événement `OnTimer` :

```
Panel1.Caption:=TimeToStr(Time);
```

Modifiez le libellé qui se trouve au-dessus du `TreeView` en Affichage d'arborescence, et le libellé qui se trouve au-dessous en Nom de fichier sélectionné.

En ce qui concerne les propriétés du composant `TreeView`, double-cliquez à droite de la propriété Items, là où figure `TTreNodes`. L'Editeur d'élément `TreeView` apparaît. Dans cet éditeur, vous pouvez ajouter les éléments et les sous-éléments qui composent votre arborescence. Ajoutez un élément appelé Niveau 1. Ajoutez ensuite un sous-élément du niveau 1 appelé Niveau 2, puis un sous-élément du niveau 2 appelé Niveau 3.

Enfin, ajoutez le code ci-après au bouton OpenDialog :

```
OpenDialog1.FileName := '*.*';
  if OpenDialog1.Execute then
    Edit1.Text := OpenDialog1.FileName;
end;
```

Lorsque vous avez fini, le code doit ressembler à celui du Listing 8.3.

Listing 8.3 : Le code source de MOREVCL

```
unit more ;
interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ExtCtrls, ComCtrls, StdCtrls, Buttons;

type
  TForm1 = class(TForm)
    Timer1: TTimer;
    Panel1: TPanel;
    OpenDialog1: TOpenDialog;
    TabControl1: TTabControl;
    TrackBar1: TTrackBar;
    ProgressBar1: TProgressBar;
    Panel2: TPanel;
    Button1: TButton;
    TreeView1: TTreeView;
    Label1: TLabel;
    Edit1: TEdit;
    Label2: TLabel;
    procedure Timer1Timer(Sender: TObject);
    procedure TrackBar1Change(Sender: TObject);
    procedure BitBtn1Click(Sender: TObject);
    procedure TabControl1Change(Sender: TObject);
    procedure Button1Click(Sender: TObject);
```

```
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Timer1Timer(Sender: TObject);
begin
     Panel1.Caption:=TimeToStr(Time);
end;

procedure TForm1.TrackBar1Change(Sender: TObject);
begin
     ProgressBar1.Position:=TrackBar1.Position;
end;

procedure TForm1.BitBtn1Click(Sender: TObject);
begin
     Application.Terminate;
end;

procedure TForm1.TabControl1Change(Sender: TObject);
begin
     If TabControl1.TabIndex=0 then Panel2.Caption:='Tab = 1';
     If TabControl1.TabIndex=1 then Panel2.Caption:='Tab = 2';
     If TabControl1.TabIndex=2 then Panel2.Caption:='Tab = 3';
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  OpenDialog1.FileName := '*.*';
  if OpenDialog1.Execute then
    Edit1.Text := OpenDialog1.FileName;
end;
```

```
end.
```

Enregistrez le programme et lancez-le. Si tout s'est bien déroulé, vous devez pouvoir déplacer la `TrackBar` et voir la `ProgressBar` suivre de pair. Vous devez pouvoir sélectionner un des trois onglets du `TabControl` et voir le titre du Panel se modifier en fonction de l'onglet sélectionné. Vous devez pouvoir également ouvrir les trois niveaux du `TreeView` en cliquant sur chaque élément. Enfin, vous devez pouvoir faire apparaître la boîte de dialogue Ouvrir en cliquant sur le bouton et en sélectionnant un nom de fichier. Le nom du fichier doit s'afficher dans la boîte d'édition. Si vous rencontrez des problèmes, revenez à votre fiche et vérifiez le code et les propriétés.

Si nous avons pu voir de nombreux composants, il a été impossible de tous les décrire. Le nombre de méthodes, de propriétés et de composants est tel qu'il serait impossible de les décrire tous en détail dans cet ouvrage. Néanmoins, cette partie doit vous avoir familiarisé avec la VCL, et il ne vous reste plus qu'à utiliser l'aide en ligne pour trouver rapidement les informations qui vous font défaut. Vous savez où chercher, et ce dont vous disposez. Vous êtes maintenant en mesure de construire des applications qui *font vraiment quelque chose*.

Récapitulatif

Dans cette partie, nous avons pu voir ensemble ce qu'était la VCL et à quoi elle servait. Nous avons vu les propriétés, les méthodes et les événements. Vous avez appris à utiliser l'aide en ligne pour trouver rapidement des informations concernant ces sujets. Nous avons fait un tour d'horizon rapide des composants qui figurent dans la VCL, OCX et composants visuels d'exemple compris. Vous avez pu construire une application, sans grande utilité sinon de faire la démonstration du fonctionnement et de l'utilisation des composants les plus usuels. Cette section a constitué pour vous un bon entraînement.

Atelier

L'atelier vous donne trois moyens de vérifier que vous avez correctement assimilé le contenu de cette partie. La section Questions - Réponses reprend des questions posées couramment, et y répond, la section Questionnaire vous pose des questions dont vous trouverez les réponses en consultant l'Annexe A, et les Exercices vous proposent de mettre en pratique ce que vous venez d'apprendre. Tentez dans la mesure du possible de vous appliquer à chacune des trois sections avant de passer au jour suivant.

Questions - Réponses

Q Quelle est la différence entre la VCL Delphi 32 bits et la VCL Delphi 16 bits ?

R Delphi 1.0 comportait 75 composants alors que Delphi 3.0 en compte plus de 130 (en comptant les composants d'exemple et les ActiveX). De plus, Delphi 3 sait utiliser les composants 32 bits.

Q Est-il possible de recompiler en 32 bits pour Delphi 3.0 une application 16 bits écrite pour Delphi 1.0 ?

R Oui. Comme les composants Delphi sont écrits en Delphi, il vous suffit de recompiler votre application avec la nouvelle version 32 bits pour passer d'une application 16 bits à une application 32 bits. La plupart des application se recompileront avec peu ou pas de modifications. Cependant, certaines applications 16 bits utilisant des VBX devront avoir recours au composant OCX ou ActiveX équivalent.

Questionnaire

1. Quelle est la différence entre un composant visuel et un composant non visuel ?
2. Qu'est-ce qu'une propriété imbriquée ?
3. Qu'est-ce qu'une méthode ?
4. Que sont les événements ?
5. Quel composant utiliser si vous souhaitez que l'utilisateur entre des données dans un format particulier ?
6. Quel composant utiliser si vous souhaitez exécuter une procédure ou une tâche toutes les cinq minutes ?

Exercices

1. Revenez dans VCLDEMO et écrivez le code pour les éléments de menu Aide et A propos (un conseil : utilisez la procédure `ShowMessage`).
2. Ecrivez une application de Réveil. Votre application doit afficher l'heure, permettre de régler une heure de sonnerie et afficher un message lorsque l'heure sélectionnée arrive (un conseil : utilisez des composants `Timer` et `Panel`, entre autres).

9

Entrée/sortie de fichiers et impression

Cette section est consacrée aux différentes tâches d'entrée-sortie de vos applications. Parmi les méthodes les plus courantes de communication, nous nous intéresserons plus particulièrement aux entrées/sorties de fichiers, y compris un certain nombre de techniques permettant de transmettre, stocker et récupérer des informations sur des fichiers disque. Vous découvrirez également comment imprimer vos œuvres.

Entrée/sortie de fichiers

Une des tâches les plus courantes et les plus précieuses en programmation est la manipulation de fichiers. Un fichier n'est rien d'autre qu'une collection ordonnée de données qui est stockée sur un disque dur, une disquette, un CD-ROM, une bande, ou tout autre support de masse. De façon évidente, les applications de bases de données doivent pouvoir créer, lire et écrire des fichiers, mais ce n'est pas le seul cas où les fichiers s'avérent indispensables. On peut ainsi utiliser des fichiers pour stocker les informations de configuration d'une application. Ils permettent de stocker temporairement des informations pour permettre à un programme d'occuper de la mémoire système précieuse à d'autres tâches, puis de recharger les informations dans la mémoire si nécessaire. Les fichiers peuvent même être utilisés pour transmettre des informations d'un programme à un autre. Enfin, bien sûr, les fichiers sont fréquemment utilisés pour enregistrer notre travail dans un traitement de textes ou un tableur. Delphi gère parfaitement les fichiers et de manière très aisée à assimiler. Le but de cette journée est de vous montrer comment travailler avec les divers types de fichiers et de vous présenter les fonctions et procédures qui simplifient les tâches liées aux fichiers.

Nous allons parler des attributs et des types de fichier. Vous apprendrez à travailler avec des fichiers texte, des fichiers de type binaires et des fichiers non typés. Vous pourrez également découvrir certaines fonctions liées aux fichiers et répertoires, fonctions qui calquent leur comportement sur des commandes DOS classiques telles que MkDir. Pour terminer, nous traiterons des noms de fichier longs.

Attributs de fichiers

Note

Cette partie suppose que vous avez une connaissance préalable des nombres binaires et de la logique booléenne. Si vous éprouvez des difficultés à en comprendre le contenu, nous vous invitons à vous documenter plus avant sur ces sujets avant de poursuivre la lecture de cet ouvrage.

Nouveau

Les fichiers ont des attributs spéciaux. Un attribut est un propriété exhibée par un fichier et qui dépend de son paramétrage. Par exemple, si la propriété lecture seule d'un fichier est activée, ce fichier peut être lu mais non mis à jour ni supprimé par la plupart des commandes ou programme DOS. Chaque fichier comporte un octet d'attribut qui stocke les paramètres d'at-

tributs. Chaque paramètre est stocké dans un des huit bits qui composent l'octet. Seuls six bits sont utilisés et, sur ces six bits utilisés, deux servent pour les répertoires et les labels de volume, ne laissant donc que quatre bits pour les attributs eux-mêmes. Ces quatre attributs sont les suivants :

Attribut de fichier	Constante Delphi	Description
`Fichiers en lecture seule`	`faReadOnly`	Permet qu'on lise le fichier, mais pas qu'on le mette à jour ou le supprime
`Fichiers cachés`	`faHidden`	Empêche que le fichier apparaisse dans une liste de répertoires normale
`Fichiers système`	`faSysFile`	Marque qu'un fichier est utilisé par le système et empêche qu'il soit visible dans une liste de répertoires normale
`Fichiers archive`	`faArchive`	Cet attribut est désactivé si un fichier a été sauvegardé
`ID Volume`	`faVolumeID`	Cet attribut sert à créer un ID de volume.
`Répertoire`	`faDirectory`	Cet attribut permet d'identifier les répertoires (dossiers)

Ces paramètres d'attributs sont activés ou désactivés par les programmes qui utilisent ou créent les fichiers. Par défaut, à l'exception du bit archive (bit 5), tous les attributs sont désactivés lors de la création. En Delphi, comme dans d'autres langages, vous pouvez modifier ces attributs avant ou après avoir travaillé sur les fichiers. Delphi propose des fonctions telles que `FileGetAttr` et `FileSetAttr` qui permettent de lire et de modifier les attributs à loisir. Ce peut être nécessaire si, par exemple, vous avez besoin de manipuler un fichier qui est en lecture seule. Il suffit alors de désactiver l'attribut lecture seule, de travailler sur le fichier puis, les modifications terminées, de réactiver l'attribut lecture seule. Il convient aussi de remarquer que l'on peut activer ou désactiver n'importe quelle combinaison de ces attributs. Comment tout cela fonctionne-t-il ? Découpons notre octet d'attributs en ses 8 composants binaires et intéressons-nous aux bits.

Les bits de l'octet attribut

Bit	Attribut stocké
`Bit 0`	Lecture Seule
`Bit 1`	Fichier Caché

Bit	Attribut stocké
`Bit 2`	Fichier Système
`Bit 3`	ID volume
`Bit 4`	Sous-répertoire
`Bit 5`	Archive
`Bit 6`	inutilisé
`Bit 7`	inutilisé

Les attributs d'un fichier tout juste créé seraient les suivants :

Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	
0	0	1	0	0	0	0	0

Soit 00100000 en binaire ou 20 en hexa. Supposons maintenant que vous souhaitiez activer la Lecture seule. Vous pouvez utiliser la fonction `FileSetAttr` de Delphi avec la constante `faReadOnly`. Cette constante est égale à 00000001 en binaire ou 01 en hexa. Lorsqu'on a fourni le nom de fichier et la valeur de `faReadOnly` à `FileSetAttr`, un OU est effectué sur la valeur et l'octet d'attribut. Dans cet exemple on désactiverait le bit d'archive, ce qui est souhaitable ou non selon ce que vous cherchez à faire. Nous verrons comment travailler avec plusieurs octets par la suite.

En examinant $AD — ou 10101101 en binaire (on lit de droite à gauche ce nombre binaire) —, vous verrez, en vous appuyant sur la table de correspondance des bits de l'octet d'attribut, qu'il correspond à un fichier dont les bits suivants sont activés : Lecture seule, Système, ID de Volume et Archive. Le bit 8 est également activé, mais il ne sert à rien. Ce n'est pas là un octet d'attribut valide. En effet, l'ID de volume et le Sous-répertoire sont des attributs que vous n'avez pas à manipuler car ils sont activés lorsque vous créez un label de volume ou un sous-répertoire. Ils n'ont pas besoin d'être modifiés par le programmeur ou l'utilisateur. Un nombre plus réaliste serait 100001 (binaire) ou $21. En consultant la même table, vous verrez que ce nombre correspond à un fichier dont les attributs Lecture seule et Archive sont activés (le fichier est en lecture seule et le fichier n'a pas été sauvegardé). Intéressons-nous maintenant à la manière dont on active ces bits. Imaginons que vous souhaitiez n'activer que le bit Lecture seule d'un fichier appelé TOTO.TXT. Vous pouvez alors placer la ligne de code suivante dans votre application :

```
FileSetAttr('TOTO.TXT',faReadOnly);
```

Dans la déclaration `FileSetAttr`, la constante `faReadOnly` est égale au binaire 00000001 ou $01. Cette valeur est copiée dans l'octet d'attribut. Dans notre exemple, le bit Lecture seul passe à 1 et tous les autres à 0.

Si vous souhaitez combiner cette constante à d'autres constantes ou nombres, il vous suffit d'effectuer des OU sur ces divers nombres. Lorsque vous effectuez un OU entre deux nombres, vous effectuez un OU entre le bit 0 du premier nombre et le bit 0 du deuxième, et ainsi de suite pour chaque bit. Si l'un des deux bits pris deux à deux est égal à 1, le résultat pour le bit est 1. Si aucun des deux bits n'est 1, le résultat est 0. Ci-après se trouve la table de vérité de la fonction OU (OR en anglais) ainsi que quelques exemples d'utilisation de OU. Dans ce cas précis, il est indifférent d'opérer des additions ou des OU, mais il n'est pas inutile de savoir ce qui se passe lorsque on effectue un OU sur deux nombres.

Table de vérité de OR (les différents résultats d'une opération OR)

```
1 OR 1 = 1
1 OR 0 = 1
0 OR 1 = 1
0 OR 0 = 0
```

Exemples :

```
10110010
OR
10010111
------------
10110111

10110101
OR
11100001
--------
11110101
```

L'autre opération avec laquelle vous devriez être familier est l'utilisation de AND (ET logique). Lorsque vous effectuez un ET entre deux bits, la valeur résultante ne vaut 1 que si les deux opérandes valent 1.

Table de vérité de AND (les différents résultats d'une opération AND)

```
1 AND 1 = 1
1 AND 0 = 0
0 AND 1 = 0
0 AND 0 = 0
```

Exemple

```
10110101
AND
10010111
------------
10010101
```

Nouveau

Dans les deux exemples de OR, vous pouvez voir que les nombres du dessus sont différents alors que les nombres du bas restent les mêmes. Les résultats sont donc bien évidemment différents. On appelle cela le masquage. Le masquage est un processus qui vous permet de lire ou de modifier sélectivement des bits dans un octet lorsque cet octet est utilisé dans une opération OR ou AND (ET) avec un octet prédéfini appelé masque. En changeant le masque (le nombre du dessus) vous pouvez activer ou désactiver sélectivement des bits. Vous pouvez ajouter des constantes ou des nombres ou opérer des OR sur eux pour créer un masque qui permettra de modifier plus d'un bit (ou attribut en l'occurrence).

Autrement dit, si vous souhaitez activer les bits Lecture seule et Archive, il vous suffit d'utiliser la déclaration : `FileSetAttr('TOTO.TXT',faReadOnly+faArchive);`

Ceci crée un masque de 00100001.

Maintenant que deux bits sont activés, comment faire pour en désactiver un seul ? Il suffit d'appeler `FileSetAttr` de nouveau et de ne lui passer qu'un masque contenant les bits que vous souhaitez laisser activés, les autres seront alors désactivés. La ligne qui suit désactive le bit d'archive tout en conservant le bit de Lecture seule :

```
FileSetAttr('TOTO.TXT',faReadOnly);
```

Si vous utilisez cet exemple, tous les bits seront remis à zéro, excepté le bit de Lecture seule. Ce n'est pas forcément ce que vous souhaitez. Ceci s'explique par le fait que la constante `faReadOnly` qui n'a qu'un bit d'activé est copiée dans l'octet d'attribut. La meilleure manière pour résoudre notre problème consiste à commencer par prendre les attributs d'un fichier pour les stocker dans une variable. Vous pouvez alors créer un masque, et il suffira d'effectuer un `OU` entre le masque et la variable (contenant les paramètres initiaux) pour ne modifier que les bits désirés. Le code pour parvenir à ce résultat est le suivant :

```
Var
     FileAttr Integer
Begin
     {on récupère le contenu de l'octet Attribut pour MYFILE.TXT}
     FileAttr:=FileGetAttr('MYFILE.TXT');
     {on effectue un OR  entre le masque (faReadOnly) et la valeur de l'octet
     d'attribut stockée dans FileAttr, et on place le résultat dans
     FileAttr}
     FileAttr:=FileAttr OR faReadOnly;
     {On donne pour valeur à l'octet d'attribut la nouvelle valeur stockée
     dans FileAttr}
```

```
    FileSetAttr('MYFILE.TXT',FileAttr);
end;
```

Voyons ce qui se passerait si in créait un nouveau fichier et qu'on le faisait passer dans la routine ci-avant. Créez un fichier appelé MONFICHIER.TXT. L'octet d'attribut pour le nouveau fichier est :

```
00100000   (Bit 5, le bit d'archive est active)
```

Récupérez l'octet d'attribut et stockez-le dans une variable appelée FileAttr. Ensuite, effectuez un `OR` entre `FileAttr` et le masque prédéfini `(faReadOnly)`, et stockez le résultat dans `FileAttr`. Vous obtenez ainsi un nouveau masque qui combine les anciens paramètres aux nouveaux.

```
00100000
    OR
00000001
     =
00100001
```

En travaillant en binaire, il est plus facile de savoir quels bits vous souhaitez activer ou désactiver, et à quoi ressemblera le résultat. Si vous convertissez les valeurs ci-avant en hexa, vous pouvez encore voir ce qui se passe. Ainsi, les nombres ci-avant, convertis en hexa, donnent :

```
$20 OR $01 = $21
```

L'octet d'attribut initial contient $20 et on effectue un `OR` entre cet octet et `faReadOnly` (qui est égal à $01), et le résultat est $21.

Supposons maintenant que vous souhaitez savoir si un bit est activé dans l'octet d'Attribut. Il faut pour cela utiliser l'opérateur `AND`. En utilisant les mêmes chiffres que précédemment, vous trouveriez qu'aucun des bits recherchés n'est activé. Cet exemple recherche le bit de Lecture seule.

```
00100000
   AND
00000001
     =
00000000
```

Toutes ces théories sur les binaires, les hexa et les attributs de fichiers sont bien, mais qu'en est-il de la pratique ? Pourquoi ne pas créer un petit programme qui vous permettra de visualiser et de définir les quatre attributs de fichiers (lecture seule, système, caché et archive) pour un fichier donné. Vous pourrez ainsi voir tous ces concepts à l'œuvre dans un utilitaire Windows.

Tout d'abord, créez un nouveau projet contenant une unique fiche et appelé FILEATTR.DPR. Le nom de l'unité sera ATTRSCR.PAS. En vous référant à la Figure 9.1, ajoutez les composants suivants à la fiche :

Quantité	Type de composant
1	Bevel
1	GroupBox
3	BitBtn
4	CheckBoxes
1	Label
1	Edit
1	DirectoryListBox
1	FileListBox
1	DriveComboBox.

Figure 9.1
La boîte de dialogue du Gestionnaire d'attributs de fichiers.

En partant du sommet, le label vient en premier, suivi par la boîte d'édition. Sous cette boîte, se trouve le DirectoryListBox à gauche et le FileListBox à droite. Plus bas on trouve le DriveComboBox. Au bas de la fiche à gauche se trouve le Bevel contenant les trois BitBtn. Enfin, en bas à droite se trouve la GroupBox contenant les quatre CheckBox. Modifiez les propriétés des composants, ajoutez un peu de code, et lancez l'application. Modifiez la propriété caption

du Label pour qu'elle devienne Nom de fichier :. Puis, supprimez tous les caractères de la propriété `Edit1.Text`.

En supposant que vous ayez ajouté les composants DirectoryListBox, FileListBox et DriveCombo, vous pouvez aisément les lier ensemble. Commencez par modifier la valeur de la propriété `FileList` du composant DirectoryListBox pour qu'elle devienne `FileListBox1`. Vous pouvez remarquer au passage que FileListBox1 était un choix que vous pouviez sélectionner. Ceci facilite grandement la connexion des trois composants. Ensuite, allez dans le composant DriveComboBox et modifiez la valeur de la propriété `DirList` pour qu'elle devienne `DirectoryListBox1`. Puis, allez dans le composant `FileListBox` et modifiez la valeur de la propriété `FileEdit` pour qu'elle devienne `Edit1`. Ces modifications font que ces composants fonctionnent de pair comme s'ils constituaient en fait un seul composant, tout comme dans une boîte de dialogue Fichier. Vous devez effectuer quelques autres modifications pour que votre programme fonctionne correctement. Si vous souhaitez que votre application soit en mesure de lire et d'afficher des fichiers cachés et système, vous devez modifier quelques propriétés dans le composant `FileListBox`. Une des propriétés de ce composant est FileType. Cette propriété contient des propriétés imbriquées que vous devez modifier. Assurez-vous que les propriétés suivantes ont pour valeur True : `ftReadOnly`, `ftSystem`, `ftHidden`, `ftArchive` et `ftNormal`. Les deux dernières propriétés (ftVolumeID et ftDirectory) doivent avoir pour valeur False. Modifiez les propriétés Caption des quatre cases à cocher : Lecture seule pour CheckBox1, Fichier caché pour CheckBox2, Système pour CheckBox2 et Archive pour CheckBox4.

Changez les noms des trois BitBtn en BitBtnOk, BitBtnSave et BitBtnClose. Dans BitBtnOk, modifiez la propriété `Kind` en `bkOk`. Pour BitBtnSave, modifiez la propriété `Kind` en `bkAll`. Enfin, pour BitBtnClose, modifiez la propriété `Kind` en `bkClose`. Sur le bouton bkSave, modifiez la propriété Caption en `&Enregistrer`. A présent, vous pouvez ajouter du code aux différents boutons.

Le code du bouton BitBtnOk doit être le suivant :

```
procedure TForm1.BitBtnOKClick(Sender: TObject);
begin
    {On récupère les attributs de fichier pour le fichier sélectionné }
    {On regarde les bits d'attributs qui sont activés et on coche les cases
    en conséquence }
    fname:=Edit1.Text;
    AttrByte:=FileGetAttr(fname);
    If AttrByte AND faReadOnly = faReadOnly Then
       CheckBox1.Checked:=True
    else
       CheckBox1.Checked:=False;
    If AttrByte AND faHidden = faHidden then
       CheckBox2.Checked:=True
    else
```

```
        CheckBox2.Checked:=False;
     If AttrByte AND faSysFile = faSysFile then
        CheckBox3.Checked:=True
     else
        CheckBox3.Checked:=False;
     If AttrByte AND faArchive = faArchive then
        CheckBox4.Checked:=True
     else
        CheckBox4.Checked:=False;
end;
```

Le code du bouton BitBtnSave doit être le suivant :

```
procedure TForm1.BitBtnSaveClick(Sender: TObject);
begin
     {on met à zéro l'octet d'attribut }
     AttrByte:=0;
     {on met à jour l'octet d'attribut en se basant sur les paramètres
indiqués par les cases à cocher }
     If CheckBox1.Checked = True then
        AttrByte:=AttrByte OR faReadOnly;
     If CheckBox2.Checked = True then
        AttrByte:=AttrByte OR faHidden;
     If CheckBox3.Checked = True then
        AttrByte:=AttrByte OR faSysFile;
     If CheckBox4.Checked = True then
        AttrByte:=AttrByte OR faArchive;
     {On écrit les paramètres des cases à cocher dans l'octet d'attribut }
     FileSetAttr(fname,AttrByte);
end;
```

Le code du bouton BitBtnClose doit être le suivant :

```
procedure TForm1.BitBtnCloseClick(Sender: TObject);
begin
     Application.Terminate;
end;
```

Vous devez par ailleurs ajouter ce qui suit à la section `implementation` :

```
Var
   fname:string;
   AttrByte:integer;
```

Le code du Gestionnaire d'événements `OnDblClick` de `FileListBox1` est le suivant :

```
procedure TForm1.FileListBox1DblClick(Sender: TObject);
begin
     Form1.BitBtnOkClick(self);
```

Si vous avez tapé correctement ce qui précède, votre code devrait ressembler à celui du Listing 9.1.

Listing 9.1 : Gestionnaire d'attributs de fichier

```
unit Attrscr;

interface

uses
 Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
Menus, StdCtrls, Buttons, FileCtrl, ExtCtrls;

type
  TForm1 = class(TForm)
    Label1: TLabel;
    Edit1: TEdit;
    GroupBox1: TGroupBox;
    CheckBox1: TCheckBox;
    CheckBox2: TCheckBox;
    CheckBox3: TCheckBox;
    CheckBox4: TCheckBox;
    BitBtnOK: TBitBtn;
    BitBtnClose: TBitBtn;
    FileListBox1: TFileListBox;
    DirectoryListBox1: TDirectoryListBox;
    DriveComboBox1: TDriveComboBox;
    BitBtnSave: TBitBtn;
    Bevel1: TBevel;
    procedure BitBtnCloseClick(Sender: TObject);
    procedure BitBtnOKClick(Sender: TObject);
    procedure BitBtnSaveClick(Sender: TObject);
    procedure FileListBox1DblClick(Sender: TObject);
  private
    { Déclarations privées }
  public
```

```
    { Déclarations publiques }
  end;

var
  Form1: TForm1;

implementation
Var
   fname:string;
   AttrByte:integer;
{$R *.DFM}

procedure TForm1.BitBtnCloseClick(Sender: TObject);
begin
     Application.Terminate;
end;

procedure TForm1.BitBtnOKClick(Sender: TObject);
begin
    {On récupère les attributs de fichier pour le fichier sélectionné }
    {On regarde les bits d'attributs qui sont activés et on coche les cases
en conséquence }
    fname:=Edit1.Text;
    AttrByte:=FileGetAttr(fname);
    If AttrByte AND faReadOnly = faReadOnly Then
       CheckBox1.Checked:=True
    else
       CheckBox1.Checked:=False;
    If AttrByte AND faHidden = faHidden then
       CheckBox2.Checked:=True
    else
       CheckBox2.Checked:=False;
    If AttrByte AND faSysFile = faSysFile then
       CheckBox3.Checked:=True
    else
       CheckBox3.Checked:=False;
    If AttrByte AND faArchive = faArchive then
       CheckBox4.Checked:=True
    else
       CheckBox4.Checked:=False;
end;
```

```
procedure TForm1.BitBtnSaveClick(Sender: TObject);
begin
     {on met à zéro l'octet d'attribut }
     AttrByte:=0;
     {on met à jour l'octet d'attribut en se basant sur les paramètres
indiqués par les cases à cocher }
     If CheckBox1.Checked = True then
        AttrByte:=AttrByte OR faReadOnly;
     If CheckBox2.Checked = True then
        AttrByte:=AttrByte OR faHidden;
     If CheckBox3.Checked = True then
        AttrByte:=AttrByte OR faSysFile;
     If CheckBox4.Checked = True then
        AttrByte:=AttrByte OR faArchive;
     {On écrit les paramètres des cases à cocher dans l'octet d'attribut }
     FileSetAttr(fname,AttrByte);
end;
procedure TForm1.FileListBox1DblClick(Sender: TObject);
begin
 BitBtnOkClick(self);
end;

end.
```

Analyse *Le code de ce Gestionnaire d'attributs de fichier comporte deux sections principales. Ces sections sont contenues dans les gestionaires `OnClick` des boutons OK (`BitBtnOKClick`) et Sauvegarder (`BitBtnSaveClick`).*

Pour tester ce programme, créez un fichier temporaire et lancez le Gestionnaire d'attributs de fichier. La case à cocher Archive doit être cochée et toutes les autres cases doivent être vides. Cochez la case Caché et cliquez sur Enregistrer. Demandez une liste de répertoire et chercher le nom du fichier. Le nom du fichier n'apparaît pas dans la liste des répertoires (à moins que vous n'utilisiez un programme qui affiche les fichiers cachés). Vous pouvez maintenant décocher la case Fichier caché. Cliquez sur Enregistrer et le fichier réapparaît dans une liste de répertoires. Passons maintenant aux types de fichiers.

Types de fichier

Il y a deux types fondamentaux de fichiers : *texte* et *binaire*. Vous pouvez stocker ou formater des données dans ces deux types de fichiers de bien des manières différentes. Comme tout, ces types ont leurs avantages et leurs inconvénients. Ce qui convient dans une situation précise ne convient pas forcément dans une autre. Regardons ensemble les différents types de fichiers et quelques exemples de leurs utilisations. Ce ne sont que des exemples. Vous êtes entièrement

libre de formater et de manipuler des données stockées dans un fichier de la manière qui vous convient.

Fichiers texte

Nouveau

Nous connaissons tous les fichiers texte. Les fichiers texte *sont des fichiers simples contenant des caractères ASCII bruts. Dans un fichier texte, les données sont généralement stockées et récupérées de manière séquentielle, une ligne à la fois. Chaque ligne se termine par des caractères retour chariot (`$D`) et saut de ligne (`$A`). Comme les données sont traitées séquentiellement, une recherche sur un grand fichier texte, ou l'apport de nombreuses modifications à un fichier texte peuvent se révéler des tâches fastidieuses et terriblement inefficaces. De manière générale, si vous comptez manipuler des données séquentiellement et que vous n'avez pas besoin d'effectuer des sauts d'un emplacement à un autre dans un même fichier, un fichier texte est parfaitement adapté à vos besoins. Nous allons examiner certaines des fonctions et procédures qui vous permettent de manipuler des fichiers texte, puis nous écrirons un programme qui stocke et lit des données dans un fichier texte.*

La première chose à considérer est le type de variable `TextFile`. `TextFile` vous permet de déclarer une variable qui permettra d'identifier le type de fichier que vous allez manipuler. Votre déclaration pourrait être la suivante :

```
Var
     MyFile : TextFile;
```

Maintenant que vous avez une variable du type `TextFile`, vous devez trouver un moyen de passer ces informations à Delphi ces informations ainsi que le nom du fichier à manipuler. Pour ce faire, vous utilisez la procédure `AssignFile`. Si vous connaissez déjà Pascal, la procédure `Assign` doit vous être familière. Delphi a une compatibilité ascendante compatible avec la procédure `Assign`, mais vous devez utiliser `AssignFile` en Delphi pour éviter des conflits d'étendue. On utilise `AssignFile` de la manière suivante :

```
AssignFile(MonFichier,NomDeFichier)
```

Où `MonFichier` est la variable que vous avez définie comme un fichier texte et NomDeFichier est une chaîne contenant le nom de fichier que vous souhaitez manipuler. Une fois que vous avez utilisé `AssignFile`, vous pouvez vous référer au fichier en employant `MonFichier`.

Vous devez aussi connaître certaines procédures permettant d'écrire dans des fichiers texte. Lisez les descriptions ci-après, puis allez voir la syntaxe et les descriptions de ces procédures dans l'aide en ligne. Etudiez avec attention les procédures qui suivent et les commentaires qui leur sont associés.

Procédure	Description
`ReWrite`	Crée et ouvre un nouveau fichier. Tout fichier existant portant le même nom sera écrasé.

Procédure	Description
`WriteLn`	Ecrit une ligne de texte dans le fichier ouvert, en ajoutant en bout de ligne une combinaison CR/LF (retour chariot/saut de ligne).
`CloseFile`	Finit la mise à jour du fichier courant et referme ce dernier.

Pour lire des fichiers texte, vous aurez besoin des procédures suivantes :

Procédure	Description
`Reset`	Ouvre un fichier existant. Les fichiers texte sont ouverts en lecture seule.
`Readln`	Lit la ligne de texte courante dans un fichier de texte ouvert. Chaque ligne s'achève par une combinaison CR/LF.

Maintenant que nous avons procédé à quelques définitions, il est temps de construire un programme qui lit et écrit dans un fichier texte simple, afin de comprendre *in vivo* le fonctionnement de ces diverses procédures. Ouvrez un nouveau fichier dans Delphi et référez-vous à la Figure 9.2 pour placer les divers composants de notre application. De haut en bas, placez Label1, Edit1, Label2 et Edit2. De gauche à droite, placez trois boutons au bas de la fiche. Changez les libellés des composants pour qu'ils correspondent à la Figure 9.2. Tout le code qu'il vous faut ajouter prend place dans ces trois boutons.

Figure 9.2
L'application de démonstration de fichiers texte.

Ajoutez ce code au bouton Enregistrer (button1) :

```
procedure TForm1.Button1Click(Sender: TObject);
Var
   OutFile : TextFile;
   fname,OutString : string;
```

```
begin
   {On affecte un nom de fichier à la variable }
   fname:='JUNKFILE.TXT';
   {on identifie le nom et le type de fichier comme OutFile}
   AssignFile(OutFile,fname);
   {on crée un nouveau fichier identifié comme OutFile}
   Rewrite(OutFile);
   {On prend le texte dans la boîte d'édition d'écriture }
   OutString:=Edit1.Text;
   {on écrit le texte de OutString dans le fichier }
   Writeln(OutFile,OutString);
   {On met à jour puis ferme le fichier }
   CloseFile(OutFile);
end;
```

Ajoutez le code ci-après au bouton Enregistrer (button2) :

```
procedure TForm1.Button2Click(Sender: TObject);
Var
   InFile : TextFile;
   fname,InString : string;
begin
   {On affecte un nom de fichier texte à la variable }
   fname:='JUNKFILE.TXT';
   {On identifie le fichier texte comme InFile}
   AssignFile(InFile,fname);
   {On ouvre le fichier identifié par InFile}
   Reset(Infile);
   {On lit une ligne de texte }
   Readln(InFile,InString);
   {On stocke la ligne lue dans la boîte Texte lu }
   Edit2.Text:=InString;
   {On ferme le fichier }
   CloseFile(InFile);
end;
```

Ajoutez le code ci-après au bouton Ferme (button3) :

```
procedure TForm1.Button3Click(Sender: TObject);
begin
     Application.Terminate;
end;
```

Le listing complet doit ressembler à celui qui figure dans le Listing 9.2.

Listing 9.2 : Textform

```
unit Textform;

interface

uses
 Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
StdCtrls;

type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Edit2: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    procedure Button3Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button3Click(Sender: TObject);
begin
     Application.Terminate;
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
Var
   OutFile : TextFile;
   fname,OutString : string;
begin
   {On affecte un nom de fichier à la variable }
   fname:='JUNKFILE.TXT';
   {on identifie le nom et le type de fichier comme OutFile}
   AssignFile(OutFile,fname);
   {on crée un nouveau fichier identifié comme OutFile}
   Rewrite(OutFile);
   {On prend le texte dans la boîte d'édition d'écriture }
   OutString:=Edit1.Text;
   {on écrit le texte de OutString dans le fichier }
   Writeln(OutFile,OutString);
   {On met à jour puis ferme le fichier }
   CloseFile(OutFile);
end;
procedure TForm1.Button2Click(Sender: TObject);
Var
   InFile : TextFile;
   fname,InString : string;
begin
   {On affecte un nom de fichier texte à la variable }
   fname:='JUNKFILE.TXT';
   {On identifie le fichier texte comme InFile}
   AssignFile(InFile,fname);
   {On ouvre le fichier identifié par InFile}
   Reset(Infile);
   {On lit une ligne de texte }
   Readln(InFile,InString);
   {On stocke la ligne lue dans la boîte Texte lu }
   Edit2.Text:=InString;
   {On ferme le fichier }
   CloseFile(InFile);
end;
end.
```

Analyse

Cette petite application toute simple vous permet d'entrer du texte dans la boîte d'édition située au sommet de la fiche. En cliquant sur le bouton Enregistrer, le texte est écrit dans un fichier appelé JUNKFILE.TXT. Vous pouvez lire ce même fichier. Le texte lu dans le fichier apparaît dans la boîte d'édition située au bas de la fiche. Faites quelques essais avec cette application. Vous pouvez même utiliser le Bloc-notes ou un autre éditeur pour éditer manuellement JUNKFILE.TXT afin de vérifier que le programme fonctionne correctement.

Nous venons de voir comment lire et écrire une ligne de texte unique, et il va sans dire que vous pouvez construire une boucle qui permette de lire ou d'écrire plusieurs lignes de texte. Un problème se pose ici : vous pouvez écrire dans un fichier jusqu'à épuiser vos ressources disque, mais comment savoir quand il faut arrêter de lire un fichier ? Autrement dit, comment savoir si vous avez rencontré la fin du fichier ? Delphi, comme d'autres langages, dispose d'une fonction qui vous indique si vous avez atteint la fin d'un fichier. Cette fonction est `Eof`. Pour utiliser `Eof`, il vous suffit de lui passer la variable de fichier utilisée avec `AssignFile` et cette fonction renvoie une valeur booléenne qui peut être utilisée dans une boucle de type `while`.

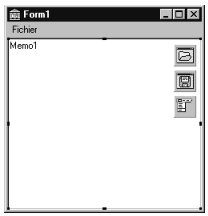
Le code ci-après est un exemple d'utilisation `d'Eof` :

```
while not Eof(InFile) do
    Begin
       Readln(InFile,MyString);
    end;
```

Cette boucle continuera jusqu'à ce qu`'Eof` renvoie `True`. Sachant cela, écrivons rapidement un petit éditeur de texte et étudions son fonctionnement. On utilisera les composants `Memo`, `OpenDialog`, `SaveDialog` et `MainMenu`. Placez un exemplaire de chacun de ces composants sur une nouvelle fiche (reportez-vous Figure 9.3 pour la disposition des composants). Cliquez sur le composant `OpenDialog` et double-cliquez sur la propriété `Filter` dans l'Inspecteur d'objets pour faire apparaître l'Editeur de filtres. Ajoutez les deux filtres suivants à l'Editeur de filtres :

Nom du filtre	Filtre
`Fichiers texte`	*.txt
`Tous`	*.*

Figure 9.3
L'application d'édition de texte WinEdit.



Faites de même dans le composant SaveDialog. En utilisant le composant MainMenu, ajoutez un menu Fichier contenant les options Ouvrir, Enregistrer, puis une ligne séparatrice et une option Quitter. Modifiez la propriété `Align` du composant `Memo` pour lui donner la valeur `alClient`. Cette valeur fera que le composant `Memo` se dimensionnera automatiquement pour remplir la totalité de la zone client de la fiche. Affectez `ssVertical` à la propriété `ScrollBar` du composant `Memo`. Il ne reste plus qu'à ajouter quelques lignes de code aux options du menu Fichier, et vous disposerez d'un éditeur de texte rudimentaire.

Ajoutez à l'option de menu Ficher/Ouvrir :

```
procedure TForm1.Open1Click(Sender: TObject);
Var
   InFile : TextFile;
   fname,InString : string;
begin
   If OpenDialog1.Execute then
   Begin
      fname:=OpenDialog1.FileName;
      AssignFile(InFile,fname);
      Reset(Infile);
      While not Eof(InFile) do
         Begin
            Readln(InFile,InString);
            Memo1.Lines.Add(InString);
         end;
      CloseFile(InFile);
      {On définit le nom de fichier en titre }
      Form1.Caption:=Form1.Caption + '['+fname+']';
   end;
end;
```

Le code ci-avant emploie une boucle pour charger une ligne de texte à la fois dans la boîte Memo jusqu'à atteindre la fin du fichier. Ce n'est pas la meilleure façon de charger du texte dans un composant Memo, mais l'objet de ce programme est de vous montrer comment fonctionne `Eof`. En fait, pour charger le fichier il vous aurait suffit de taper la ligne suivante :

```
Memo1.Lines.LoadfromFile(OpenDialog1.FileName);
```

Cette méthode donnerait ce qui suit :

```
begin
  if OpenDialog1.Execute then
  begin
    Memo1.Lines.LoadfromFile(OpenDialog1.FileName);
  end;
```

A l'option de menu Fichier/Enregistrer, ajoutez :

```
procedure TForm1.Save1Click(Sender: TObject);
Var
   OutFile : TextFile;
   fname : string;
begin
   If SaveDialog1.Execute then
     begin
        fname:=SaveDialog1.FileName;
        AssignFile(OutFile,fname);
        Rewrite(OutFile);
        {On écrit le contenu du memo en un bloc }
        Writeln(OutFile,Memo1.Text);
     end;
   CloseFile(OutFile);
   {On définit le nom de fichier du titre }
   Form1.Caption:=Form1.Caption + '['+fname+']';

```

Le code de l'option de menu Fichier/Enregistrer écrit le texte dans la boîte mémo sous la forme d'un seul bloc de texte, il est donc inutile d'utiliser une boucle. Il peut arriver que vous ayez besoin d'une boucle pour écrire ligne par ligne du texte dans un fichier. Le principe est similaire à la lecture ligne à ligne, à ceci près que vous devez ici connaître le nombre de lignes à écrire ou disposer d'une méthode pour déterminer le moment où la boucle se terminera.

Enfin, dans l'option de menu Fichier/Quitter, ajoutez :

```
procedure TForm1.Exit1Click(Sender: TObject);
begin
     Application.Terminate;
end;
```

Le Listing 9.3 montre le code complet de notre Editeur.

Listing 9.3 : Edit

```
unit Edit;

interface

uses
 Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
StdCtrls, Menus;
```

```
type
  TForm1 = class(TForm)
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Open1: TMenuItem;
    Save1: TMenuItem;
    N1: TMenuItem;
    Exit1: TMenuItem;
    Memo1: TMemo;
    OpenDialog1: TOpenDialog;
    SaveDialog1: TSaveDialog;
    procedure Open1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure Exit1Click(Sender: TObject);
    procedure Save1Click(Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Open1Click(Sender: TObject);
Var
   InFile : TextFile;
   fname,InString : string;
begin
   If OpenDialog1.Execute then
   Begin
      fname:=OpenDialog1.FileName;
      AssignFile(InFile,fname);
      Reset(Infile);
      While not Eof(InFile) do
         Begin
```

```
            Readln(InFile,InString);
            Memo1.Lines.Add(InString);
         end;
      CloseFile(InFile);
      {On définit le nom de fichier en titre }
      Form1.Caption:=Form1.Caption + '['+fname+']';
   end;
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
     {Texte en clair dans la boîte mémo}
     Memo1.Text:='';
end;

procedure TForm1.Exit1Click(Sender: TObject);
begin
     Application.Terminate;
end;

procedure TForm1.Save1Click(Sender: TObject);
Var
   OutFile : TextFile;
   fname : string;
begin
   If SaveDialog1.Execute then
     begin
        fname:=SaveDialog1.FileName;
        AssignFile(OutFile,fname);
        Rewrite(OutFile);
        {On écrit le contenu du memo en un bloc }
        Writeln(OutFile,Memo1.Text);
     end;
   CloseFile(OutFile);
   {On définit le nom de fichier du titre }
   Form1.Caption:=Form1.Caption + '['+fname+']';
end;
end.
```

Analyse

Ce programme ressemble aux éditeurs de texte Windows standard tels que le Bloc-notes, même s'il offre beaucoup moins de fonctions. Si vous ne l'avez pas déjà fait, faites Fichier/Enregistrer projet sous et enregistrez l'unité sous EDIT.PAS et le fichier de projet sans WINEDIT.DPR. Lancez le programme et essayez de charger un fichier texte. Apportez quelques modifications au fichier et enregistrez-le sous un nouveau nom. Utilisez le Bloc-notes ou un autre éditeur pour créer vos fichiers de texte et pour vérifier la validité des fichiers créés ou mis à jour par votre éditeur.

Comme nous l'avons dit précédemment, les façons de formater et de manipuler les fichiers texte sont légion. Soyez imaginatif et rappelez-vous que tout problème admet des solutions multiples. Vous devez déterminer vous-même ce qui convient le mieux pour la tâche précise que vous avez à mener.

Fichiers binaires

Le deuxième type de fichier est le fichier binaire. *Tous les fichiers de type non texte entrent dans cette catégorie. Un fichier binaire n'est rien d'autre qu'un fichier contenant des informations binaires écrites par un programme. Dans le cas des caractères ASCII, le code ASCII représente les informations binaires écrites dans le fichier. A la différence des textes fichier, tout fichier ouvert comme fichier binaire y compris les textes, fichiers de programme, bitmap, etc., peut être lu par votre programme. Dans ce mode, c'est à vous qu'il incombe de déterminer la façon de traiter les données que vous pouvez lire.*

Il existe deux catégories de fichiers binaires : les fichiers typés et les fichiers non typés. Ces catégories sont décrites dans les sections suivantes.

Fichiers typés

Le *fichier typé* est l'un des différents types de fichiers binaires. Ce sont des fichiers dont vous avez choisi le format ou la structure, ainsi que le type (et la longueur) des données que vous y stockez, que ce soit des entiers, des réels, des chaînes, etc. Un bon exemple de fichier typé est un fichier utilisé par un programme de répertoire téléphonique. Créons un programme simple qui montre l'intérêt d'un tel fichier. Créez un nouveau projet, en enregistrant l'unité sous ADDR.PAS et le projet sous ADDRESS.DPR. Commencez par créer votre structure d'enregistrements. Dans la partie d'implémentation, placez le code ci-après juste sous la directive {$R *.DFM} :

```
type
  Address = record
       Lastname: String[20];
       Firstname: String[20];
       Phone: String[15];
       StreetAddress : String[50];
       City : String[40];
       State : String[2];
```

```
        ZipCode : String[10];
    end;
```

Vous venez de créer votre propre type appelé Address. Notre exemple n'utilisera que des chaînes, mais vous auriez pu utiliser n'importe quel autre type de variable (integer, byte, real, etc.) dans votre type d'enregistrement. Utilisez ce nouveau type pour créer des variables de votre programme. Ensuite, juste au-dessous du code précédent, ajoutez ceci :

```
Var
   AddressFile : File of Address;
   AddressData : Address;
```

`AddressFile` est une variable Fichier de type `Adress`. Les variables Fichier sont utilisées dans la plupart des procédures et fonctions d'E/S fichier. Comme votre nouveau type, `Address`, est un enregistrement, `AddressData` devient un tampon contenant la structure du type `Address`. Ceci vous permet de lire ou d'écrire facilement les données dans le tampon en une seule ligne de code, tout en faisant respecter un certain ordre. Tous les enregistrements enregistrés sur le disque auront la même taille, quelles que soient les données qui y figurent. Il s'agit là d'un *enregistrement de taille fixe*, notion implicite dans tous les fichiers typés.

Avant de nous enfoncer dans les profondeurs de notre programme, examinons ensemble les procédures et fonctions qui permettent de manipuler les fichiers typés. Pour plus de détails sur ces fonctions et procédures, reportez-vous à la documentation Delphi ou à l'aide en ligne. Il ne s'agit ici que de vous présenter les outils.

`AssignFile`

Syntaxe : `procedure AssignFile(var F : File, Chemin : String);`

Utilité : Permet d'affecter un nom de fichier à une variable de fichier à l'attention d'autres fonctions d'E/S de fichier.

`Reset`

Syntaxe : `procedure Reset(var F : File[; RecSize: Word] );`

Utilité : Permet d'ouvrir un fichier existant qui a été affecté à une variable de fichier au moyen de `AssignFile`.

`Rewrite`

Syntaxe : `procedure Rewrite (var F : File[; RecSize: Word] );`

Utilité : Permet de créer et d'ouvrir un fichier existant qui a été affecté à une variable de fichier au moyen de `AssignFile`.

`Seek`

Syntaxe : `procedure Seek (var F : File; N : Longint );`

Utilité : Permet de déplacer le pointeur de fichier jusqu'à l'enregistrement spécifié (par N) dans le fichier ouvert.

`Read`

Syntaxe : `procedure Read(F , V1 [, V2,...,Vn ] );`

Utilité : Permet de lire des enregistrements dans un fichier.

`Write`

Syntaxe : `procedure Write (F , V1 [, V2,...,Vn ] );`

Utilité : Permet d'écrire des enregistrements dans un fichier.

`Eof`

Syntaxe : `function Eof(var F): Boolean;`

Utilité : Permet de déterminer si le programme a atteint la fin du fichier. Utilisé en conjonction avec `Read`.

`CloseFile`

Syntaxe : `procedure CloseFile(var F);`

Utilité : Permet de mettre à jour les modifications du fichier et de refermer ce dernier.

Exemple de projet

Continuons notre programme. Vous avez déjà créé le projet, ajouté un type appelé Address et quelques variables. Ajoutez ces variables juste au-dessous de la ligne `AddressDate : Address;`

```
Fname : String;
RecSize,CurRec : Longint;
```

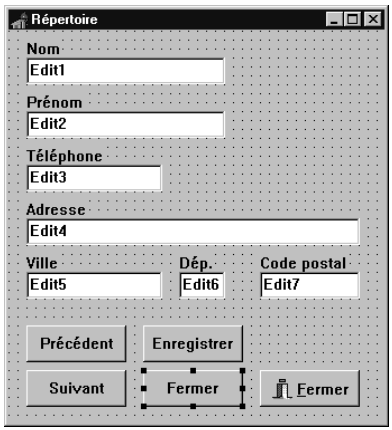
Ajoutez maintenant sept boîtes d'édition, sept libellés, quatre boutons et un BitBtn. En vous référant à la Figure 9.4, disposez les boîtes d'édition et les libellés, en commençant par placer Edit1 et Label1 au sommet. Modifiez les propriétés caption et name des boutons et celles du BitBtn.

Le Listing 9.4 sera créé par Delphi. Examinez ce code et utilisez-le pour ajouter le code adéquat aux divers composants.

Listing 9.4 : Addr

```
unit Addr;

interface

uses
 Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
StdCtrls, Buttons;

```

Figure 9.4
Exemple d'utilisation de fichiers typés : un programme de répertoire d'adresses.




```
type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    Edit4: TEdit;
    Edit5: TEdit;
    Edit6: TEdit;
    Edit7: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    Label7: TLabel;
    Previous: TButton;
    Next: TButton;
    Save: TButton;
    New: TButton;
    Close: TBitBtn;
    procedure FormCreate(Sender: TObject);
    procedure NewClick(Sender: TObject);
    procedure PreviousClick(Sender: TObject);
    procedure NextClick(Sender: TObject);
```

```
    procedure SaveClick(Sender: TObject);
    procedure CloseClick(Sender: TObject);
  private
    { Déclarations privées }
    procedure LoadRecord;
    procedure SaveRecord;
    procedure ShowRecord;
    procedure ClearData;
  public
    { Déclarations publiques }

  end;

var
  Form1: TForm1;

implementation
{$R *.DFM}

type
  Address = record
       Lastname: String[20];
       Firstname: String[20];
       Phone: String[15];
       StreetAddress : String[50];
       City : String[40];
       State : String[2];
       ZipCode : String[10];
  end;
Var
   AddressFile : File of Address;
   AddressData : Address;
   Fname : String;
   RecSize,CurRec : Longint;

procedure TForm1.LoadRecord;
begin
  {on charge l'enregistrement }
  Read(AddressFile,AddressData);
  {on affiche l'enregistrement à l'écran }
  ShowRecord;
```

```
end;

procedure TForm1.SaveRecord;
begin
  {On copie l'enregistrement de l'écran vers l'enregistrement }
  AddressData.Lastname:=Edit1.Text;
  AddressData.Firstname:=Edit2.Text;
  AddressData.Phone:=Edit3.Text;
  AddressData.StreetAddress:=Edit4.Text;
  AddressData.City:=Edit5.Text;
  AddressData.State:=Edit6.Text;
  AddressData.ZipCode:=Edit7.Text;
  {On écrit l'enregistrement sur le disque }
  Write(AddressFile,AddressData);
end;

procedure TForm1.ClearData;
begin
       {On met à zéro les boîtes d'édition }
       Edit1.Text:='';
       Edit2.Text:='';
       Edit3.Text:='';
       Edit4.Text:='';
       Edit5.Text:='';
       Edit6.Text:='';
       Edit7.Text:='';
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
   {On met à zéro les boîtes d'édition }
   ClearData;
   {On met à zéro le compteur d'enregistrement courant }
   CurRec:=0;
   {On définit le nom de fichier }
   Fname:='ADDRESS.DAT';
   {On définit la variable de fichier }
   AssignFile(AddressFile,Fname);
   {On regarde la taille de l'enregistrement }
   RecSize:=SizeOf(AddressData);
   {Si le fichier existe, on le charge }
   If FileExists(Fname) then
```

```
   Begin
       Reset(AddressFile);
       If not Eof(AddressFile) then
          begin
             Read(AddressFile,AddressData);
             ShowRecord;
          end;
   end

   {Sinon, on le crée }
   else
       Begin
          ClearData;
          Rewrite(AddressFile);
       end;
end;

procedure TForm1.NewClick(Sender: TObject);
begin
    repeat
       CurRec:=CurRec+1;
       Seek(AddressFile,CurRec);
    until Eof(AddressFile);
    {On met à zéro les boîtes d'édition }
    ClearData;
    {On crée un nouvel enregistrement }
    SaveRecord;
    {On revient à l'enregistrement courant }
    Seek(AddressFile,CurRec);
end;

procedure TForm1.PreviousClick(Sender: TObject);
begin
     If CurRec-1 < 0 then
     begin
     {Si on est avant le début du fichier, on se place au premier
      enregistrement et on affiche le message }
         CurRec:=0;
         Seek(AddressFile,CurRec);
         ShowMessage('Ceci est le début du fichier');
     end
```

```
     {Sinon, on recule d'un enregistrement et on affiche }
     else
     Begin
         CurRec:=CurRec-1;
         Seek(AddressFile,CurRec);
         Read(AddressFile,AddressData);
         Seek(AddressFile,CurRec);
         ShowRecord;
     end;
end;

procedure TForm1.NextClick(Sender: TObject);
begin
    {On passe à l'enregistrement suivant }
    CurRec:=CurRec+1;
    Seek(AddressFile,CurRec);
    {Si on est avant la fin du fichier, on lit l'enregistrement et on affiche }
    If not Eof(AddressFile) Then
       begin
         Read(AddressFile,AddressData);
         Seek(AddressFile,CurRec);
         ShowRecord;
       end
    {Sinon, on revient au dernier enregistrement et on affiche le message }
    else
       begin
         CurRec:=CurRec-1;
         Seek(AddressFile,CurRec);
         ShowMessage('Ceci est la fin du fichier');
       end;
end;

procedure TForm1.ShowRecord;
begin
  {On copie les données d'enregistrement dans les boîtes d'édition }
  Form1.Edit1.Text:=AddressData.Lastname;
  Form1.Edit2.Text:=AddressData.Firstname;
  Form1.Edit3.Text:=AddressData.Phone;
  Form1.Edit4.Text:=AddressData.StreetAddress;
  Form1.Edit5.Text:=AddressData.City;
  Form1.Edit6.Text:=AddressData.State;
```

```
  Form1.Edit7.Text:=AddressData.ZipCode;
end;

procedure TForm1.SaveClick(Sender: TObject);
begin
     {On sauvegarde l'enregistrement}
     SaveRecord;
     {On affiche l'enregistrement dans des boîtes d'édition }
     ShowRecord;
end;

procedure TForm1.CloseClick(Sender: TObject);
begin
     {On enregistre l'enregistrement courant }
     SaveRecord;
     {On ferme le fichier }
     CloseFile(AddressFile);
     {On quitte l'application }
     Application.Terminate;
end;

end.
```

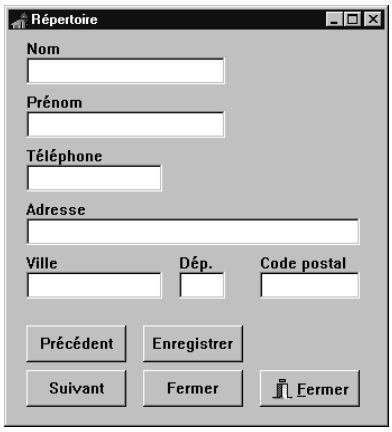
Analyse

Vous venez d'écrire une application de base de données d'adresses sans utiliser de moteur de base de données. Ce programme crée un fichier appelé address.dat, prend ce que saisit l'utilisateur et le stocke dans le fichier sous forme d'enregistrements. Le code du bouton Nouveau efface les champs de saisie et permet à l'utilisateur d'entrer un nouvel enregistrement. Le bouton Enregistrer écrit les données dans le fichier address.dat. Les boutons Précédent et Suivant vous permettent de naviguer dans la base de données enregistrement par enregistrement. Enfin, le bouton Fermer ferme l'application.

Il est temps de tester notre application. Lancez-la. Vous devriez voir apparaître l'écran de la Figure 9.5.

Lorsque le programme est lancé pour la première fois, il regarde si le fichier de données est présent. Dans notre cas, il ne trouve pas de fichier et le crée (ADDRESS.DAT). Entrez votre nom, votre adresse et votre numéro de téléphone dans les boîtes d'édition et sélectionnez Enregistrer. Pour entrer l'enregistrement suivant, cliquez sur le bouton Nouveau. Les boîtes sont vidées et vous pouvez entrer l'enregistrement suivant. Entrez quelques autres enregistrements en procédant de la même façon. Testez ensuite l'application en vous déplaçant dans les enregistrements à l'aide des boutons Précédent et Suivant. Vous pourrez remarquer que lorsque vous arrivez au début ou à la fin du fichier, une boîte de message apparaît indiquant que vous avez atteint la fin de l'enregistrement. Il est important d'empêcher des erreurs ou de les captu-

Figure 9.5
Exemple utilisant des fichiers typés : programme de répertoire.



rer, et d'empêcher également que l'utilisateur dépasse le début ou la fin du fichier. Sortez du programme et lancez-le à nouveau. Cette fois, le programme trouve le fichier de données (ADDRESS.DAT), le charge et en affiche le premier enregistrement. Comme auparavant, vous devez pouvoir vous déplacer dans les enregistrements, les éditer, en enregistrer et en ajouter de nouveaux.

Maintenant que vous avez créé une application fonctionnelle, prenez un peu de temps pour étudier le code. Lisez les commentaires du listing pour comprendre ce qui se passe. Dans l'événement `FormCreate`, remarquez la présence du code qui regarde si le fichier de données existe. Si c'est le cas, il est ouvert à l'aide de `Reset`. Le premier enregistrement est alors chargé puis affiché à l'écran. Sinon, le fichier de données est créé à l'aide de `Rewrite` et un écran vierge s'affiche, prêt pour la saisie. Etudiez également le code et les commentaires associés aux boutons et aux procédures `LoadRecord`, `SaveRecord`, `ShowRecord`, `ClearData`. Lorsque vous sentez que vous avez bien assimilé les fichiers typés, passez à la partie suivante.

Fichiers non typés

Les *fichiers non typés* vous permettent de manipuler les fichiers avec plus de souplesse. Vous pouvez aller en n'importe quel emplacement du fichier, modifier un octet ou un bloc entier, enregistrer les données et fermer le fichier. Lorsque vous écrivez votre code, vous n'avez pas à vous conformer à des structures rigides et votre code peut traiter n'importe quel type de fichiers, de n'importe quelle façon. Il y a un inconvénient cependant. En effet, vous devez écrire votre code en déterminant très exactement à quel endroit du fichier vous souhaitez travailler. Vous devez pour cela utiliser des pointeurs de fichier, vous appuyer sur une bonne connaissance du fichier lui-même et mettre en œuvre des algorithmes pointus dans votre application. Les tailles des enregistrements peuvent varier, et il incombe au programmeur de déterminer où se trouve un enregistrement de données, et quelle est sa taille. La plus grande prudence est de mise lorsque vous travaillez avec des fichiers non typés, mais le jeu en vaut parfois la chandelle. Imaginez que vous ayez un fichier de taille conséquente dans lequel vous souhaitez remplacer

tous les espaces par des virgules. Vous pouvez écrire un programme Delphi simple qui transférera le fichier dans un tampon, bloc par bloc, puis cherchera dans le tampon les espaces et les transformera en virgules au fur et à mesure avant d'enregistrer les modifications sur le disque. Il est indifférent que le fichier soit un fichier texte ou binaire, de même que la façon dont sont stockées les données dans ce fichier n'importe pas. Pour tout dire, vous êtes totalement libre dans le choix de votre méthode d'accès ou de manipulation des données dans un fichier non typé. Avant de poursuivre sur ce sujet, examinons quelques procédures et fonctions qui vous seront utiles.

`BlockRead`

Syntaxe : `BlockRead(var F: File; var Buf; Count: Word [; var Result: Word]);`

Utilité : Lit un bloc de données sur le disque et le place dans un tampon.

`BlockWrite`

Syntaxe : `BlockWrite(var f: File; var Buf; Count: Word [; var Result: Word]);`

Utilité : Permet d'écrire un bloc de données de la mémoire vers le disque.

`FilePos`

Syntaxe : `FilePos(var F): Longint;`

Utilité : Permet de récupérer la position courante du pointeur de fichier.

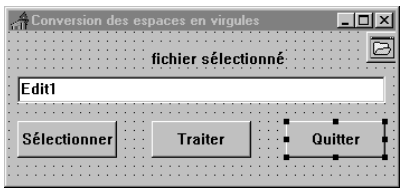
Exemple de projet

Créons un projet qui lit un fichier dans lequel il convertira tous les espaces en virgules.

Suivez ces étapes et créez la fiche pour l'application.

1. Créez un nouveau projet Delphi et enregistrez l'unité sous le nom de SP2CMA.PAS.
2. Enregistrez le projet sous SP2COMMA.DPR.
3. Ajoutez trois boutons, une boîte d'édition et un label à la fiche. La fiche doit ressembler à celle de la Figure 9.6.

Figure 9.6
Programme de conversion des espaces en virgules.



4. Modifiez le libellé de la fiche pour qu'il devienne "Conversion des espaces en virgules".
5. Ajoutez un composant OpenDialog à la fiche.

6. Double-cliquez sur la propriété `Filter` et utilisez l'Editeur de filtres pour donner comme nom de filtre "Tous" et comme filtre "*.*".
7. Définissez comme False les propriétés `Visible` de la boîte d'édition et du label.

Le cœur du programme réside dans l'événement `OnClick` du bouton Action. C'est là que se trouve le code qui charge les données provenant du disque, puis convertit les espaces en virgules et enregistre les données modifiées sur le disque. Regardons de plus près certains aspects de ce code.

Dans la partie qui suit, vous définissez quelques variables nécessaires à l'opération. `InFile` est une variable de type `File` dont nous avons déjà parlé. Fbuffer est un tableau de type `Byte` qui est un tampon de 1 ko permettant de stocker les données lues dans un fichier. Fpointer permet de stocker le pointeur de fichier (la position dans le fichier) afin de pouvoir revenir à cette position après une opération de lecture ou d'écriture.

```
Var
   InFile : File;
   FBuffer : array [0..1023] of Byte;
   FPointer : Longint;
   BytesRead : Integer;
   x : Integer;
```

Vous devez ensuite identifier le fichier et le préparer au traitement. Vous avez déjà rencontré `AssignFile`, mais vous pouvez remarquer qu'on a ajouté ici un paramètre à la fonction `Reset`. Ce paramètre représente la taille d'enregistrement, qui est de 128 octets par défaut. Donnez-lui une valeur de 1 pour notre exemple.

```
begin
AssignFile(InFile,Fname);
Reset(InFile,1);
```

Une fois le fichier ouvert, vous devez le charger par morceaux de 1 ko. La boucle while se prolonge tant que vous n'avez pas atteint la fin du fichier. A chaque passage dans la boucle, vous obtenez la position du pointeur de fichier. Vous lisez un bloc à l'aide de la procédure `BlockRead` qui place 1 ko de données dans le tampon. Ensuite, vous parcourez les données du buffer, octet par octet, en transformant en virgules tous les espaces rencontrés. Une fois le tampon traité, vous utilisez la procédure Seek pour placer le pointeur de fichier au début de l'enregistrement. Ceci est nécessaire car le pointeur de fichier est déplacé jusqu'au début du bloc suivant de données après une opération de lecture ou d'écriture. Vous utilisez ensuite `BlockWrite` pour enregistrer sur disque le contenu du tampon. S'il reste des données à traiter, vous repartez dans la boucle, sinon le fichier est mis à jour puis fermé à l'aide de `CloseFile`. Lorsque vous avez fini, un message s'affiche indiquant Traitement achevé.

```
unit Addr;

interface

uses
 Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
StdCtrls, Buttons;

type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    Edit4: TEdit;
    Edit5: TEdit;
    Edit6: TEdit;
    Edit7: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    Label7: TLabel;
    Previous: TButton;
    Next: TButton;
    Save: TButton;
    New: TButton;
    Close: TBitBtn;
    procedure FormCreate(Sender: TObject);
    procedure NewClick(Sender: TObject);
    procedure PreviousClick(Sender: TObject);
    procedure NextClick(Sender: TObject);
    procedure SaveClick(Sender: TObject);
    procedure CloseClick(Sender: TObject);
  private
    { Déclarations privées }
    procedure LoadRecord;
    procedure SaveRecord;
    procedure ShowRecord;
    procedure ClearData;
```

```
  public
    { Déclarations publiques }

  end;

var
  Form1: TForm1;

implementation
{$R *.DFM}

type
  Address = record
       Lastname: String[20];
       Firstname: String[20];
       Phone: String[15];
       StreetAddress : String[50];
       City : String[40];
       State : String[2];
       ZipCode : String[10];
  end;
Var
   AddressFile : File of Address;
   AddressData : Address;
   Fname : String;
   RecSize,CurRec : Longint;

procedure TForm1.LoadRecord;
begin
  {on charge l'enregistrement }
  Read(AddressFile,AddressData);
  {on affiche l'enregistrement à l'écran }
  ShowRecord;
end;

procedure TForm1.SaveRecord;
begin
  {On copie l'enregistrement de l'écran vers l'enregistrement }
  AddressData.Lastname:=Edit1.Text;
  AddressData.Firstname:=Edit2.Text;
  AddressData.Phone:=Edit3.Text;
```

```
  AddressData.StreetAddress:=Edit4.Text;
  AddressData.City:=Edit5.Text;
  AddressData.State:=Edit6.Text;
  AddressData.ZipCode:=Edit7.Text;
  {On écrit l'enregistrement sur le disque }
  Write(AddressFile,AddressData);
end;

procedure TForm1.ClearData;
begin
       {On met à zéro les boîtes d'édition }
       Edit1.Text:='';
       Edit2.Text:='';
       Edit3.Text:='';
       Edit4.Text:='';
       Edit5.Text:='';
       Edit6.Text:='';
       Edit7.Text:='';
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
   {On met à zéro les boîtes d'édition }
   ClearData;
   {On met à zéro le compteur d'enregistrement courant }
   CurRec:=0;
   {On définit le nom de fichier }
   Fname:='ADDRESS.DAT';
   {On définit la variable de fichier }
   AssignFile(AddressFile,Fname);
   {On regarde la taille de l'enregistrement }
   RecSize:=SizeOf(AddressData);
   {Si le fichier existe, on le charge }
   If FileExists(Fname) then
   Begin
       Reset(AddressFile);
       If not Eof(AddressFile) then
          begin
             Read(AddressFile,AddressData);
             ShowRecord;
          end;
   end
```

```
   {Sinon, on le crée }
   else
       Begin
          ClearData;
          Rewrite(AddressFile);
       end;
end;

procedure TForm1.NewClick(Sender: TObject);
begin
    repeat
       CurRec:=CurRec+1;
       Seek(AddressFile,CurRec);
    until Eof(AddressFile);
    {On met à zéro les boîtes d'édition }
    ClearData;
    {On crée un nouvel enregistrement }
    SaveRecord;
    {On revient à l'enregistrement courant }
    Seek(AddressFile,CurRec);
end;

procedure TForm1.PreviousClick(Sender: TObject);
begin
     If CurRec-1 < 0 then
     begin
     {Si on est avant le début du fichier, on se place au premier
enregistrement et on affiche le message }
         CurRec:=0;
         Seek(AddressFile,CurRec);
         ShowMessage('Ceci est le début du fichier');
     end
     {Sinon, on recule d'un enregistrement et on affiche }
     else
     Begin
         CurRec:=CurRec-1;
         Seek(AddressFile,CurRec);
         Read(AddressFile,AddressData);
         Seek(AddressFile,CurRec);
         ShowRecord;
```

```
     end;
end;

procedure TForm1.NextClick(Sender: TObject);
begin
    {On passe à l'enregistrement suivant }
    CurRec:=CurRec+1;
    Seek(AddressFile,CurRec);
   {Si on est avant la fin du fichier, on lit l'enregistrement et on affiche }
    If not Eof(AddressFile) Then
       begin
         Read(AddressFile,AddressData);
         Seek(AddressFile,CurRec);
         ShowRecord;
       end
    {Sinon, on revient au dernier enregistrement et on affiche le message }
    else
       begin
         CurRec:=CurRec-1;
         Seek(AddressFile,CurRec);
         ShowMessage('Ceci est la fin du fichier');
       end;
end;

procedure TForm1.ShowRecord;
begin
  {On copie les données d'enregistrement dans les boîtes d'édition }
  Form1.Edit1.Text:=AddressData.Lastname;
  Form1.Edit2.Text:=AddressData.Firstname;
  Form1.Edit3.Text:=AddressData.Phone;
  Form1.Edit4.Text:=AddressData.StreetAddress;
  Form1.Edit5.Text:=AddressData.City;
  Form1.Edit6.Text:=AddressData.State;
  Form1.Edit7.Text:=AddressData.ZipCode;
end;

procedure TForm1.SaveClick(Sender: TObject);
begin
SaveRecord;
     {On affiche l'enregistrement dans des boîtes d'édition }
     ShowRecord;
```

```
end;

procedure TForm1.CloseClick(Sender: TObject);
begin
     {On enregistre l'enregistrement courant }
     SaveRecord;
     {On ferme le fichier }
     CloseFile(AddressFile);
     {On quitte l'application }
     Application.Terminate;
end;

end.
```

Le code ci-avant sera rapide pour deux raisons. Tout d'abord, les procédures `BlockRead` et `BlockWrite` chargent un bloc de données tout entier en une seule opération. Par ailleurs, le traitement se fait en mémoire (dans le tampon) ce qui est beaucoup plus rapide que de devoir lire et écrire en progressant octet par octet. La taille du tampon peut être définie à une valeur plus grande si vous souhaitez améliorer encore plus les performances (pour les fichiers de taille importante principalement).

Le Listing 9.5 montre la totalité du code. Etudiez-le et prenez-le comme référence pour placer le code dans les boutons restants. Lorsque vous avez terminé, testez votre application.

Listing 9.5 : Sp2cma

```
unit Sp2cma;

interface

uses
 Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
Menus, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    OpenDialog1: TOpenDialog;
    Edit1: TEdit;
    Label1: TLabel;
```

```
    procedure Button2Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  Form1: TForm1;

implementation
Var
   Fname : String;
{$R *.DFM}
procedure TForm1.Button2Click(Sender: TObject);
Var
   InFile : File;
   FBuffer : array [0..1023] of Byte;
   FPointer : Longint;
   BytesRead : Integer;
   x : Integer;
begin
{On définit le nom de fichier comme variable de fichier }
AssignFile(InFile,Fname);
{La taille d'enregistrement est d'un octet }
Reset(InFile,1);
While Not Eof(InFile) Do
      Begin
           {On regarde la position dans le fichier }
           FPointer:=FilePos(InFile);
           {On lit un bloc de 1 ko dans le tampon }
           BlockRead(InFile,FBuffer,SizeOf(FBuffer),BytesRead);
           {on convertit les espaces en virgules }
           For x:= 0 to BytesRead-1 do
           Begin
                If FBuffer[x]=32 then FBuffer[x]:=44;
           end;
           {On replace Filepointer au début du bloc }
           Seek(InFile,FPointer);
```

```
            {On écrit le tampon de 1 ko sur le disque }
            BlockWrite(InFile,FBuffer,BytesRead);
       end;
       {On vide les tampons sur le disque et on ferme le fichier }
       CloseFile(InFile);
       ShowMessage('Traitement achevé');
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
  {On affiche la boîte de dialogue Ouvrir }
  OpenDialog1.FileName := '*.*';
  if OpenDialog1.Execute then
    {On prend les noms de fichiers sélectionnés }
    Fname := OpenDialog1.FileName;
    {On les affiche }
    Edit1.Text:=Fname;
    {Maintenant que l'on dispose du nom de fichier, on affiche la boîte Edit
et son label }
    Edit1.Visible:=True;
    Label1.Visible:=True;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
     {On quitte l'application }
     Application.Terminate;
end;

end.
```

Analyse

Ce programme lit un fichier texte (sélectionné par l'utilisateur) dans fBuffer par blocs de 1 024 octets. Le contenu du tampon est alors scruté et les espaces sont convertis en virgules (lorsque l'utilisateur appuie sur le bouton Traitement). Le pointeur de fichier est repositionné au début du bloc (car il a été avancé à la suite de la déclaration `BlockRead`) et les données modifiées sont écrites dans le fichier. A la fin du traitement, le programme affiche le message Traitement achevé. L'utilisateur peut alors traiter un autre fichier ou sélectionner Quitter pour fermer le programme.

Enregistrez le code, puis compilez-le et exécutez-le. Si tout a l'air de bien se comporter, allez dans le Bloc-notes et créez un fichier de texte simple contenant une phrase ou deux et qui vous servira de cobaye. Revenez à l'application et cliquez sur le bouton Sélection fichier. Dans la boîte de dialogue qui apparaît alors, sélectionnez le fichier texte que vous avez créé. Cliquez

ensuite sur le bouton Traitement. Une boîte de message devrait très vite apparaître, vous indiquant que le programme a terminé sa tâche. Sortez du programme et revenez dans le Bloc-notes pour charger de nouveau le fichier texte. Si le code ne comportait pas d'erreurs, vous devez voir des virgules là où se trouvaient auparavant des espaces. Vous pouvez également tester le programme sur d'autres fichiers. Pour vérifier que tout se passe correctement avec un fichier binaire, utilisez un gros document écrit dans un traitement de textes tel que Write ou Word 6, qui enregistrent leurs fichiers dans un format binaire. Ceci termine notre introduction aux fichiers non typés, même si nous n'avons fait qu'effleurer toutes leurs potentialités. A vous de jouer...

Gestion de fichiers, gestion de répertoires et autres fonctions liées aux fichiers

La liste des fonctions et procédures d'E/S et de gestion de fichier à connaître n'est pas close. Cet ouvrage ne suffirait pas pour toutes les présenter. Tâchez de vous familiariser aux fonctions et procédures qui figurent dans l'aide en ligne dans les trois rubriques suivantes :

- Routines de Gestion de fichier.
- Routines d'E/S.
- Routines de fichiers texte.

Vous avez déjà eu l'occasion d'utiliser bon nombre des fonctions et procédures qui figurent dans ces rubriques, mais d'autres doivent vous être inconnues. La plupart des fonctions effectuent les mêmes types d'opérations mais de différentes manières. Ainsi :

```
Var
    MyFileVar : File
Begin
    AssignFile(MyFileVar,filename);
    ReWrite(MyFile);
End;
```

La routine ci-avant créer une variable Fichier et lui affecte un nom de fichier. La variable `Fichier` est utilisée par la procédure `Rewrite` pour créer et ouvrir le fichier. Vous pouvez parvenir au même résultat en utilisant la fonction `FileCreate`, comme le montre le code ci-après :

```
Var
   Handle : Integer;
Begin
   Handle :=FileCreate('Filename');
End;
```

Nouveau

La fonction `FileCreate` *renvoie un* handle *de fichier si l'opération est réussie. Un handle de fichier n'est rien d'autre qu'une valeur entière qui permettra d'identifier le fichier jusqu'à sa fermeture. Si plus d'un fichier est ouvert, à chaque fichier sera affecté un handle qui lui est propre. Cet handle de fichier est utilisé par de nombreuses fonctions et procédures pour lire, écrire, déplacer le pointeur de fichier, etc., comme le faisaient les procédures et fonctions que nous avons vues, mais qui, elles, utilisaient des variables de fichier.*

Utiliser des handles plutôt que des variables de fichier présente certains avantages. Ainsi, la fonction `FileOpen` vous permet d'effectuer un `OR` sur un ensemble de constantes pour définir le mode dans lequel le fichier sera ouvert. Les constantes de mode de fichier sont indiquées ci-après :

- `fmOpenRead`
- `fmOpenWrite`
- `fmOpenReadWrite`
- `fmShareCompat`
- `fmShareExclusive`
- `fmShareDenyWrite`
- `fmShareDenyRead`
- `fmShareDenyNone`

Pour plus de détails sur ces constantes, vous pouvez vous reporter à l'aide en ligne. Vous trouverez leur description dans l'aide consacrée à l'unité SysUtils. Si vous souhaitez ouvrir un fichier avec un accès exclusif et empêcher qu'un autre utilisateur ou programme puisse y accéder, il vous suffit d'utiliser la constante `fmShareExclusive`. Par exemple :

```
MyHandle:=FileOpen(fname,fmShareExclusive);
```

N'oubliez pas que vous pouvez effectuer un `OR` sur ces constantes pour définir à votre guise les modalités d'accès au fichier. Il peut devenir nécessaire de le faire si vous partagez des fichiers avec d'autres programmes dans le même système, ou avec d'autres utilisateurs dans un réseau.

Dans le tableau ci-après figurent d'autres fonctions et routines dont vous devez connaître l'existence. Prenez le temps de les regarder et de consulter l'aide en ligne à leur sujet.

`Erase`	Supprime un fichier
`FileSize`	Donne la taille du fichier spécifié
`GetDir`	Donne le répertoire courant du lecteur spécifié
`MkDir`	Crée un sous-répertoire
`Rename`	Renomme un fichier
`RmDir`	Supprime un sous-répertoire

De nombreuses fonctions et procédures ayant trait à la manipulation des fichiers n'ont pas pu être abordées ici. Si vous avez lu l'aide en ligne concernant les trois rubriques citées plus haut, vous avez pu compléter les connaissances acquises au cours de cette journée.

Noms longs de fichiers

Nous en arrivons enfin aux noms longs ! Vous n'êtes plus limité à des noms de fichiers de 11 caractères (8 pour le nom de fichier, 3 pour l'extension). Sous Windows 95 et NT, les noms de fichiers peuvent occuper jusqu'à 255 caractères, y compris le zéro terminal. La taille maximum d'un chemin est de 260 caractères, y compris le zéro terminal.

Pour des raisons de compatibilité ascendante, un nom court est également créé à partir des six premiers caractères du nom long. Si plusieurs noms longs ont en commun les six premiers caractères, le système d'exploitation utilise un algorithme incrémental de nommage des fichiers en format 8.3. Voici quelques exemples de noms longs avec leur équivalent de nom court :

Nom long	Nom court
`LongFichierNuméro1.TXT`	LONGFI~1.TXT
`LongFichierNuméro2.TXT`	LONGFI~2.TXT

Delphi 3 sait tirer parti des noms longs. Vous n'avez rien de particulier à faire, Delphi et le système d'exploitation gèrent tout ceci de façon transparente pour vous. Il vous suffit d'utiliser le nom long dans vos fonctions, procédures et programmes. Vous pouvez également continuer d'utilser l'ancien format 8.3, la compatibilité est assurée.

Imprimer

Bien que nous vivions dans un monde de plus en plus électronique, dans lequel nous utilisons télécopie, messages électroniques et logiciels de présentation, il vient toujours un moment où il est nécessaire d'imprimer du texte ou un graphique venant d'un programme, pour générer des formulaires ou des brochures par exemple.

Nous n'utiliserons pas QuickReport dans cette section (reportez-vous au Jour 12 pour en savoir plus sur QuickReport). Nous verrons deux méthodes pour imprimer directement à partir de programmes Delphi, ce qui peut faire gagner beaucoup de temps si vous n'avez pas besoin de toutes les fonctionnalités de QuickReport ou d'autres logiciels d'impression. Nous parlerons des techniques d'impression de base, consistant à envoyer une ligne ou une chaîne de texte à la fois vers l'imprimante, comme c'est le cas dans un programme DOS simple en Pascal. Cette section examine également les objets imprimante disponibles dans Delphi. A l'aide de ces objets, vous pouvez imprimer du texte. Vous verrez comment utiliser les boîtes de dialogue d'Impression et comment imprimer des graphiques. Nous aborderons également toutes les

bases vous permettant de procéder à des impressions dans vos applications sans avoir besoin d'utiliser pour cela des produits extérieurs.

Bases de l'impression en Pascal

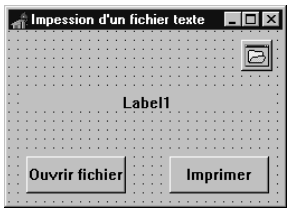
Si vous connaissez déjà les techniques d'impression en Pascal ou dans d'autres langages, vous ne serez pas surpris. L'impression dans sa plus simple expression consiste à créer une variable de fichier et à l'affecter à l'imprimante. Vous utilisez alors une déclaration `writeln` pour envoyer le texte vers l'imprimante. Ce type d'impression est des plus primitifs comparé aux fonctionnalités dont vous disposez dans Windows, mais il suffit parfois amplement. Imaginez par exemple qu'un ordinateur est connecté à une imprimante texte qui permet d'obtenir des sorties papier des mesures d'un instrument. Ou que vous souhaitez imprimer une liste simple, sans avoir besoin d'utiliser de graphiques, de polices et sans formatage particulier. Le code ci-après utilise une déclaration writeln pour imprimer :

```
var
  P : TextFile;
begin
   AssignPrn(P);
   rewrite(P);
   writeln(P,'Test d''impression');
   -CloseFile(P);
end;
```

Comme vous pouvez le voir, on déclare une variable `P` de type `TextFile`. On utilise ici une variante de `Assign`, `AssignPrn`. Cette fonction affecte la variable au port de l'imprimante, le traitant comme un fichier. Il faut ensuite ouvrir le port de l'imprimante et on utilise `rewrite` à cet effet. Le texte est envoyé à l'imprimante par le biais de la procédure `writeln` et le port de l'imprimante est fermé avec `CloseFile`. Il est important de fermer le port de l'imprimante pour terminer l'opération. Tout texte restant encore en mémoire est envoyé vers l'imprimante et le port est fermé, tout comme un fichier.

Pour comprendre le fonctionnement de cette méthode, créons un petit programme qui sélectionne un fichier texte et l'envoie à l'imprimante. Ajoutez à une fiche deux boutons, un composant `OpenDialog` et un libellé. Utilisez la propriété `filter` de `l'OpenDialog` pour ajouter un filtre .txt. Référez-vous à la Figure 9.7 pour concevoir l'écran. Puis recopiez le code qui lui est associé (Listing 9.6). Enregistrez le projet sous FILE2PRN.DPR et l'unité sous FILEPRN.PAS.

Figure 9.7
Impression de fichier texte : FILE2PRN.DPR.



Note *Vous devez ajouter* `Printers` *à la clause* `uses` *pour pouvoir accéder aux fonctions d'impression.*

Listing 9.6 : Impression d'un fichier texte

```
unit Fileprn;

interface

uses
 Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
StdCtrls,Printers;

type
  TForm1 = class(TForm)
    Button1: TButton;
    OpenDialog1: TOpenDialog;
    Button2: TButton;
    Label1: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
```

```
Var
   Fname : String;

procedure TForm1.Button1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    Fname:=OpenDialog1.FileName;
    Label1.Caption:='Prêt à imprimer ' + Fname;
  end;
end;

procedure TForm1.Button2Click(Sender: TObject);
Var
   P,F : TextFile;
   TempStr : String;
begin
   AssignFile(F,Fname);
   Reset;
   AssignPrn(P);
   Rewrite(P);
   Label1.Caption:='Impression de '+ Fname;
   While Not Eof Do
   begin
        Readln(F,TempStr);
        Writeln(P,TempStr);
   end;
   CloseFile;
   CloseFile(P);
   Label1.Caption:='Impression achevée';
end;

end.
```

Analyse

Ce programme vous permet d'utiliser la boîte de dialogue Ouvrir fichier pour choisir le fichier à imprimer. Il lit le fichier ligne par ligne et envoie chaque ligne à l'imprimante. Testez ce programme en imprimant quelques fichiers texte.

Imprimer avec l'objet Tprinter de Delphi

La place nous manque pour décrire en détail toutes les propriétés et méthodes des objets imprimante. Dans cette partie, nous développerons quelques programmes donnant des exemples d'utilisation des objets imprimante. Vous verrez ainsi comment doter vos applications de fonctionnalités d'impression.

En Delphi, vous utilisez l'objet `Tprinter` pour accéder à l'interface d'impression Windows. L'unité `Printers` de Delphi contient la variable `Printer` qui est déclarée comme instance de l'objet `Tprinter` :

```
Printer : Tprinter;
```

Pour utiliser l'objet `TPrinter`, vous devez ajouter l'unité `Printers` à la clause `uses` de votre code. A la différence d'autres unités usuelles, `Printers` n'est pas ajouté d'office par Delphi.

```
uses
 Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
StdCtrls,Printers;
```

Cela fait, vous pouvez utiliser `Printer` pour référencer des propriétés dans l'objet `TPrinter`.

L'objet TPrinter

Avant de pouvoir utiliser l'objet TPrinter, vous devez connaître certaines de ses propriétés et méthodes :

`Canvas`	Déclarée comme une instance de l'objet `TCanvas`. Le canevas est l'endroit où la page ou le document est construit en mémoire avant d'être imprimé. Ses propriétés Pen et Brush vous permettent d'y dessiner et d'y placer du texte.
`TextOut`	Méthode de l'objet TCanvas qui permet d'envoyer du texte vers le canevas.
`BeginDoc`	Permet de lancer une tâche d'impression.
`EndDoc`	Permet de terminer une tâche d'impression. L'impression proprement dite ne commence pas tant que EndDoc n'a pas été appelé.
`PageHeight`	Provoque un saut de page sur l'imprimante et redéfinit à (0,0) la valeur de la propriété Pen du canevas.
`PageNumber`	Renvoie le numéro de la page actuellement imprimée.

Ainsi, si vous souhaitez imprimer du texte à l'aide de l'objet Printer, vous écrirez quelque chose comme :

```
Printer.BeginDoc;
Printer.Canvas.TextOut(10,10,'J''imprime avec l''objet Printer');
Printer.EndDoc;
```

Ce code provoque l'impression du texte `J'imprime avec l'objet Printer` à partir du dixième pixel à droite et du dixième pixel en partant du haut du canevas. `BeginDoc` lance l'impression. Le texte est envoyé au canevas à l'aide de la propriété `TextOut` du canevas. `EndDoc` déclenche l'impression du texte proprement dite et termine la tâche d'impression.

Ces propriétés et méthodes ne sont que la partie émergée de l'iceberg `Tprint`, mais elles sont suffisantes pour créer le programme d'impression de fichier que nous avons écrit plus haut. Chargez le projet FILE2PRN.DPR et enregistrez-le sous FILE2POB.DRP. Enregistrez l'unité sous FILEPOBJ.PAS. Le programme lit le fichier ligne à ligne comme auparavant, mais il calcule aussi la position du canevas qui recevra le texte, avant de placer ce texte sur le canevas pour l'imprimer. Le Listing 9.7 montre la version améliorée du programme Impression de fichier. Les commentaires vous aideront à comprendre le code. Si vous avez fait une copie de FILEOBJ.PAS, éditez le code du bouton Imprimer (Button2 en l'occurrence) en vous référant au Listing 9.7. Vous pouvez modifier la propriété `Caption` pour qu'elle devienne `Impression de fichier texte à l'aide de l'objet TPrinter`.

Listing 9.7 : Impression de fichiers texte à l'aide de l'objet TPrinter

```
unit Filepobj;

interface

uses
 Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
StdCtrls,Printers;

type
  TForm1 = class(TForm)
    Button1: TButton;
    OpenDialog1: TOpenDialog;
    Button2: TButton;
    Label1: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  Form1: TForm1;
```

```
implementation

{$R *.DFM}
Var
   Fname : String;

procedure TForm1.Button1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    Fname:=OpenDialog1.FileName;
    Label1.Caption:='Prêt à imprimer ' + Fname;
  end;
end;

procedure TForm1.Button2Click(Sender: TObject);
Var
   F : TextFile;
   TempStr,PageNum : String;
   Ctr,x,PHeight,LineSpace: Integer;
begin
   Ctr:=1;
   {On ouvre le fichier à imprimer }
   AssignFile(F,Fname);
   Reset;
   {On lance l'impression }
   Printer.BeginDoc;
   {On récupère la hauteur de la page }
   PHeight:=Printer.PageHeight;
   {On calcule la valeur de l'interligne en se basant sur une page de 60
lignes }
   LineSpace:=PHeight DIV 60;
   {On récupère la page à imprimer }
   PageNum:=IntToStr(Printer.PageNumber);
   {On met à jour le label avec le numéro de page courant }
   Label1.Caption:='Impression de '+ Fname + ' Page ' + PageNum;
   While Not Eof Do
   begin
   {On lit une ligne de texte du fichier texte dans TempStr}
   Readln(F,TempStr);
   {On envoie le contenu de TempStr à l'imprimante }
```

```
   Printer.Canvas.TextOut(0,x,TempStr);
   {On incrémente x du nombre de pixels adéquat pour imprimer la ligne
suivante}
   x:=x+LineSpace;
   {On compte le nombre de lignes imprimées }
   Ctr:=Ctr+1;
   {Si on a imprimé 60 lignes, on commence une nouvelle page, on
récupère le numéro de page et on reprend tout }
   If Ctr > 59 then
   begin
            Printer.NewPage;
            x:=0;
            Ctr:=0;
            PageNum:=IntToStr(Printer.PageNumber);
            Label1.Caption:='Impression de '+ Fname + ' Page ' + PageNum;
   end;
   end;
   {On ferme le fichier texte et on fait s'imprimer la tâche }
   CloseFile;
   Printer.EndDoc;
   Label1.Caption:='Impression achevée.' + ' Pages imprimées = '+ PageNum;
end;

end.
```

Analyse *Ce programme se comporte comme auparavant, mais il effectue maintenant les impressions en passant par l'objet printer. Les messages d'état ont également été légèrement modifiés.*

Assurez-vous de bien enregistrer votre travail avant de tester cette nouvelle version. Le programme se comporte comme avant, à cela près qu'il imprime désormais par le biais de l'objet `Printer`. Les messages d'état ont également été modifiés.

Cette méthode d'impression vous permet d'ajouter facilement des graphiques à un document. Vous pouvez également ajouter du code pour modifier l'orientation de la page, la taille des polices, les styles et bien d'autres aspects encore. Vous bénéficiez de toutes ces fonctionnalités sans devoir écrire beaucoup plus de code. Il suffit d'indiquer à l'objet Printer ce que vous souhaitez imprimer et comment vous souhaitez que ce soit imprimé.

Les composants TPrinterDialog et TPrinterSetupDialog

Vous avez sans doute vu des logiciels du commerce qui utilisent des boîtes de dialogue pour sélectionner des options d'impression telles que le nombre de copies, le rassemblement de copies et l'orientation de la page. Il n'est pas difficile avec Delphi d'obtenir ces fonctions. Il vous suffit de placer le composant `TPrinterDialog` sur la page et d'ajouter quelques lignes de

code. Votre application dispose alors de boîtes de dialogue Impression. Le code qui le permet est le suivant :

```
if PrintDialog1.Execute then
   Begin
       {votre code d'impression}
  end;
```

Lorsque ce code est exécuté, la boîte de dialogue Impression standard de Windows apparaît, permettant à l'utilisateur de sélectionner des options pour la tâche d'impression en attente. Ces sélections faites, la tâche d'impression est menée à bien en tenant compte des paramètres spécifiés (sauf pour le nombre de copies, pour lequel il est nécessaire de créer une boucle). Ce n'est pas plus compliqué que ça.

Pour voir comment tout cela fonctionne, modifions notre programme d'impression pour qu'il comporte une boîte de dialogue Impression. Enregistrez FILE2POB.DPR sous PRINTDLG.DPR et FILE2POB.PAS sous PRNDLG.PAS. Dans l'onglet Dialogues, sélectionnez le composant `PrintDialog` et déposez-le sur la fiche. Référez-vous au Listing 9.8 pour modifier le code du bouton Imprimer. Et voilà, vous pouvez tester votre nouveau programme.

Listing 9.8 : Ajout d'une boîte de dialogue Imprimer

```
Var
   F : TextFile;
   TempStr,PageNum : String;
   Ctr,x,PHeight,LineSpace: Integer;
begin
   Ctr:=1;
   {On ouvre le fichier texte à imprimer }
   AssignFile(F,Fname);
   Reset;
   if PrintDialog1.Execute then
   Begin
     Printer.BeginDoc;
     PHeight:=Printer.PageHeight;
     LineSpace:=PHeight DIV 60;
     PageNum:=IntToStr(Printer.PageNumber);
     Label1.Caption:='Impression de '+ Fname + ' Page ' + PageNum;
     While Not Eof Do
     begin
        Readln(F,TempStr);
        Printer.Canvas.TextOut(0,x,TempStr);
        x:=x+LineSpace;
```

```
        Ctr:=Ctr+1;
        If Ctr > 59 then
        begin
            Printer.NewPage;
            x:=0;
            Ctr:=0;
            PageNum:=IntToStr(Printer.PageNumber);
            Label1.Caption:='Impression de '+ Fname + ' Page ' + PageNum;
        end;
      end;
     CloseFile;
     Printer.EndDoc;
     Label1.Caption:='Impression achevée!' + ' Pages imprimées = ' + PageNum;
  end;
end;
```

Analyse

Les seules différences visibles entre le programme précédent et notre nouveau programme sont les déclarations `if PrintDialog` *et* `begin...end` *:*

```
if PrintDialog1.Execute then
begin
        {code d'impression...}
end;
```

Lorsque vous lancez le programme et sélectionnez Imprimer, la boîte de dialogue Impression apparaît (voir Figure 9.8). Elle permet à l'utilisateur d'effectuer ses choix avant de lancer l'impression. Vous pouvez aller dans la boîte de dialogue Propriétés en cliquant sur le bouton Propriétés (voir Figure 9.9).

Figure 9.8
La boîte de dialogue Impression.

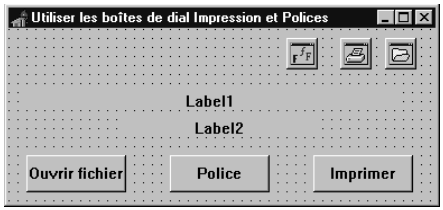
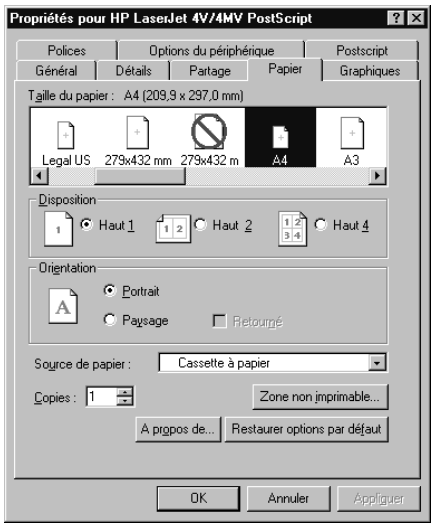


Figure 9.9
La boîte de dialogue Propriétés.



Polices et tailles de police

Il est tout aussi facile de changer de police et de taille de police. Supposons que vous souhaitez que la taille de la police passe à 18 et que la police devienne Times New Roman. Il vous suffit alors d'utiliser la propriété `Font` du canevas. En ajoutant la ligne de code ci-après à la routine d'impression de votre programme, vous pouvez choisir une police conforme à vos besoins :

```
{On définit la taille de police }
Printer.Canvas.Font.Size:=18;
{On définit la police comme Times Roman}
Printer.Canvas.Font.Name:='Times New Roman';
```

Cependant, un problème se pose. Le programme initial se fondait sur la police système 10 points par défaut et supposait qu'il y avait 60 lignes par page. Lorsque vous changez de police et de taille de police, notre ancien code ne fonctionnera plus parfaitement. Pour remédier à ce problème, ajoutons quelques lignes de code qui calculent le nombre de lignes par page en fonction de la taille de la police. Le code ci-après, par exemple, ajoute 10 lignes à la taille de la police et calcule la variable `LinesPerPage` pour ménager un espace entre les lignes lors du calcul de `LineSpace` :

```
{On calcule le nb de lignes par page en fonction de la taille de police }
     LinesPerPage:=PHeight Div FontSize+10;
     {On calcule l'interligne en fonction du nb de lignes par page }
     LineSpace:=PHeight DIV LinesPerPage;
```

Les variables que nous avons ajoutées pour prendre en compte les changements sont `FontSize` et `LinesPerPage`. Ajoutez ces lignes au code du bouton Imprimer. Assurez-vous bien que le code résultant ressemble à celui du Listing 9.9.

Listing 9.9 : Le code du bouton Imprimer

```
procedure TForm1.Button2Click(Sender: TObject);
Var
   F : TextFile;
   TempStr,PageNum : String;
   Ctr,x,PHeight,LineSpace,LinesPerPage,FontSize: Integer;
begin
   Ctr:=1;
   {On ouvre le fichier texte à imprimer }
   AssignFile(F,Fname);
   Reset;
   if PrintDialog1.Execute then
   Begin
     {On commence l'impression }
     Printer.BeginDoc;
     {On récupère la taille de la page }
     PHeight:=Printer.PageHeight;
     {Taille de police à 18}
     Printer.Canvas.Font.Size:=18;
     {Type de polices à Times Roman}
     Printer.Canvas.Font.Name:='Times New Roman';
     {On calcule le nb de lignes par page en fonction de la police, en
ajoutant 10 pour laisser un peu d'espace entre les lignes }
     LinesPerPage:=PHeight Div FontSize+10;
     {On calcule l'interligne en fonction du nb de lignes par page }
     LineSpace:=PHeight DIV LinesPerPage;
     {numéro de la page à imprimer }
     PageNum:=IntToStr(Printer.PageNumber);
     {On met à jour le label avec le numéro de page courant }
     Label1.Caption:='Impression de '+ Fname + ' Page ' + PageNum;
     While Not Eof Do
     begin
   {On lit une ligne de texte du fichier texte dans TempStr}
   Readln(F,TempStr);
   {On envoie le contenu de TempStr à l'imprimante }
   Printer.Canvas.TextOut(0,x,TempStr);
   {On ajoute à x le nb de pixels nécessaire à l'impression de la ligne
```

```
suivante }
   x:=x+LineSpace;
   {Compte du nb de lignes imprimées }
   Ctr:=Ctr+1;
   {Si on a imprimé LinesPerPage lignes, on commence une nouvelle page,
on récupère le no de page et on recommence }
   If Ctr > LinesPerPage-1 then
   begin
            Printer.NewPage;
            x:=0;
            Ctr:=0;
            PageNum:=IntToStr(Printer.PageNumber);
            Label1.Caption:='Impression de '+ Fname + ' Page ' + PageNum;
   end;
     end;
    {On ferme le fichier texte et on imprime la tâche }
    CloseFile;
    Printer.EndDoc;
    Label1.Caption:='Impression achevée!' + ' Pages imprimées = '+ PageNum;
  end;
end;
```

Analyse

Ce programme imprime des fichiers texte en Times New Roman et en corps 18, calcule le nombre de lignes de texte qui peuvent être imprimées par page et commence une nouvelle page en conséquence.

Cette technique vous permet de laisser l'utilisateur choisir la police au moyen d'une boîte de dialogue Polices. Les informations de police proviennent de l'utilisateur et vous ne les codez plus "en dur" dans votre programme. Nous allons mettre une dernière touche à notre programme en ajoutant la boîte de dialogue Police pour que l'utilisateur puisse sélectionner la police, la taille de la police, le style de la police, etc. Ajoutez un bouton Police. Nous allons également ajouter du code pour que le programme imprime le nombre de copie demandé par l'utilisateur.

Pour modifier les paramètres de police, vous pouvez utiliser la propriété `Fonte` et ses propriétés pour obtenir des informations provenant de la boîte de dialogue Police afin de les appliquer au canevas.`Printer.Canvas.Font.Size:=FontDialog1.Font.Size`; définit la taille de la police. `Printer.Canvas.Font.Name:='Times New Roman`; définit la police comme étant du Times New Roman. `Printer.Canvas.Font.Type:=FontDialog1.Font.Type`; définit le style (gras, italique...) donné par la boîte de dialogue Police.

Pour le nombre de copies, vous devez créer une boucle qui enserre le code d'impression et l'exécute un nombre de fois égal au nombre de copies demandé par l'utilisateur. Pour obtenir le nombre de copies demandé par l'utilisateur, on utilise cette déclaration : `NumCopies:=PrintDialog1`. Le nombre de copies est stocké dans la variable `NumCopies`, pour un usage ultérieur.

Pour modifier notre programme, éditez le code du bouton Imprimer pour qu'il ressemble à celui du Listing 9.10. Ajoutez un deuxième libellé à la fiche, juste au-dessus de Label1. Examinez le code de près, car certaines lignes ont été déplacées pour que la boucle fonctionne. Les commentaires vous seront précieux.

Listing 9.10 : Le bouton Imprimer contenant le code pour Police et Nombre de copies

```
procedure TForm1.Button2Click(Sender: TObject);
Var
   F : TextFile;
   TempStr,PageNum : String;
   Ctr,x,PHeight,LineSpace,LinesPerPage,FontSize,CopyNum,NumCopies:
Integer;
begin
   if PrintDialog1.Execute then
   Begin
     Ctr:=1;
     AssignFile(F,Fname);
     NumCopies:=PrintDialog1.Copies;
     for CopyNum:=1 to NumCopies do
     begin
       {Ouvre le fichier au début }
       Reset;
       {remet les compteurs à zéro pour la passe suivante }
       x:=0;Ctr:=0;
       {début de l'impression }
       Printer.BeginDoc;
       PHeight:=Printer.PageHeight;
       Printer.Canvas.Font.Size:=FontDialog1.Font.Size;
       {Définit le nom de la police pris dans la boîte de dialogue Police}
       Printer.Canvas.Font.Name:=FontDialog1.Font.Name;
      {Définit le style de la police pris dans la boîte de dialogue Police }
       Printer.Canvas.Font.Style:=FontDialog1.Font.Style;
       {Calcul du nb de lignes par pages}
       LinesPerPage:=PHeight Div FontSize+10;
       {Calcul de l'interligne}
       LineSpace:=PHeight DIV LinesPerPage;
       {Numéro de la page à imprimer }
       PageNum:=IntToStr(Printer.PageNumber);
       {Mise à jour du label }
       Label1.Caption:='Impression de '+ Fname + ' Page ' + PageNum;
       While Not Eof Do
```

```
          begin
           Readln(F,TempStr);
           Printer.Canvas.TextOut(0,x,TempStr);
           x:=x+LineSpace;
           Ctr:=Ctr+1;
           If Ctr > LinesPerPage-1 then
           begin
               Printer.NewPage;
               x:=0;
               Ctr:=0;
               PageNum:=IntToStr(Printer.PageNumber);
               Label1.Caption:='Impression de '+ Fname + ' Page ' + PageNum;
           end;
          end;
            CloseFile;
        Printer.EndDoc;
        Label1.Caption:='impression achevée!' + ' Pages imprimées = '+
        PageNum;
        Label2.Caption:='Nombre de Copies = ' + IntToStr(NumCopies);
      end;
    end;
end;
```

Analyse

Ces modifications prennent en compte les paramètres de la boîte de dialogue Police (voir Figure 9.10) et les utilisent pour la tâche d'impression. Cette tâche d'impression est exécutée le nombre de fois spécifié par l'utilisateur dans la boîte de dialogue Imprimer.

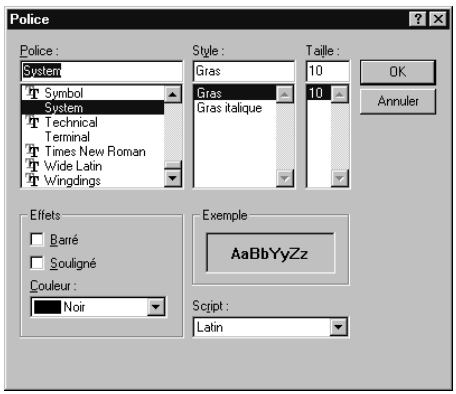


Figure 9.10
La boîte de dialogue Police.

Vous pouvez en savoir plus sur les propriétés disponibles en vous reportant à l'aide en ligne. Lorsque vous aurez apporté toutes ces modifications au programme, il devrait ressembler à celui qui figure dans le Listing 9.11.

Listing 9.11 : Les boîtes de dialogue Police et Impression

```
unit Prndlg;

interface

uses
 Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
StdCtrls,Printers;

type
  TForm1 = class(TForm)
    Button1: TButton;
    OpenDialog1: TOpenDialog;
    Button2: TButton;
    Label1: TLabel;
    PrintDialog1: TPrintDialog;
    FontDialog1: TFontDialog;
    Button3: TButton;
    Label2: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
Var
   Fname : String;
```

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    Fname:=OpenDialog1.FileName;
    Label1.Caption:='Ready to print ' + Fname;
  end;
end;

procedure TForm1.Button2Click(Sender: TObject);
Var
   F : TextFile;
   TempStr,PageNum : String;
   Ctr,x,PHeight,LineSpace,LinesPerPage,FontSize,CopyNum,NumCopies:
Integer;
begin
   if PrintDialog1.Execute then
   Begin
     Ctr:=1;
     {Ouverture du fichier à imprimer }
     AssignFile(F,Fname);
     {Nombre de copies à imprimer pris dans la boîte de dialogue Impression }
     NumCopies:=PrintDialog1.Copies;
     {Boucle pour chaque copie à imprimer }
     for CopyNum:=1 to NumCopies do
     begin
       {Ouverture du fichier au début }
       Reset;
       {Remise à zéro des compteurs pour la passe suivante }
       x:=0;Ctr:=0;
       {Début de l'impression }
       Printer.BeginDoc;
       {Hauteur de page }
       PHeight:=Printer.PageHeight;
       {Taille de police prise dans la boîte de dialogue Police }
       Printer.Canvas.Font.Size:=FontDialog1.Font.Size;
       {Nom de la police }
       Printer.Canvas.Font.Name:=FontDialog1.Font.Name;
       {Style de la police }
       Printer.Canvas.Font.Style:=FontDialog1.Font.Style;
       {calcul du nb de lignes par page }
```

```
        LinesPerPage:=PHeight Div FontSize+10;
        {Calcul de l'interligne }
        LineSpace:=PHeight DIV LinesPerPage;
        {numéro de la page imprimée }
        PageNum:=IntToStr(Printer.PageNumber);
        {mise à jour du label }
        Label1.Caption:='Impression de '+ Fname + ' Page ' + PageNum;
        While Not Eof Do
          begin
           Readln(F,TempStr);
           Printer.Canvas.TextOut(0,x,TempStr);
           x:=x+LineSpace;
           Ctr:=Ctr+1;
           If Ctr > LinesPerPage-1 then
           begin
               Printer.NewPage;
               x:=0;
               Ctr:=0;
               PageNum:=IntToStr(Printer.PageNumber);
               Label1.Caption:='Impression de '+ Fname + ' Page ' + PageNum;
           end;
          end;
        CloseFile;
        Printer.EndDoc;
        Label1.Caption:='Impression achevée!' + ' Pages imprimées = '+
        PageNum;
        Label2.Caption:='Number of Copies = ' + IntToStr(NumCopies);
      end;
    end;
end;
procedure TForm1.Button3Click(Sender: TObject);
begin
     FontDialog1.Execute;
end;

end.
```

Analyse

Le code du Listing 9.11 permet à l'utilisateur de sélectionner un fichier à imprimer par le biais d'une boîte de dialogue Ouvrir fichier de Windows. Le bouton Police permet la sélection par l'utilisateur de la police par le biais de la boîte de dialogue Police de Windows. Enfin, le bouton Imprimer fait apparaître la boîte de dialogue Impression de Windows, donnant ainsi

accès à toutes les fonctionnalités qui y figurent. Comme vous pouvez le voir, la plus grande partie du code se trouve dans l'événement `OnClick` du bouton Imprimer.

La fiche du programme définitif apparaît Figure 9.11.

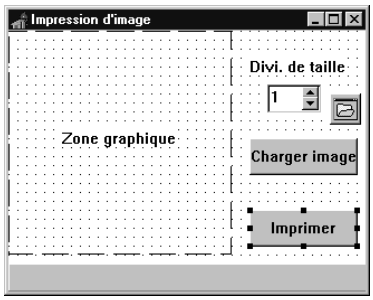


Figure 9.11
Le programme d'impression avec boîtes de dialogue Impression et Police.

Imprimer des graphiques

Vous avez imprimé du texte en utilisant la méthode traditionnelle et vous avez également envoyé du texte par le biais de l'objet `TPrinter`. Mais qu'en est-il des représentations graphiques ? Vous désirez peut-être créer des logos, des graphiques et d'autres informations non textuelles. Dans cette partie, nous verrons comment imprimer pratiquement n'importe quel graphique. Vous n'êtes en fait limité que par l'imprimante que vous utilisez.

Envoyer des graphiques à l'imprimante n'est pas très différent d'envoyer des graphiques à l'écran. Vous utilisez la propriété `Canvas` de l'objet `TPrinter` et ses propriétés et méthodes pour dessiner ou disposer des graphiques sur le canevas. Vous pouvez en fait concevoir des graphiques en commençant par les envoyer à l'écran. Lorsque vous aurez une bonne idée de leur aspect, il vous suffira de modifier le code pour envoyer le graphique à l'imprimante.

Prenons un exemple. Le code ci-après trace un cercle dans le coin supérieur gauche de la fiche `Form1` :

```
begin
     {Définit une épaisseur de crayon de 5 pixels }
     Form1.Canvas.Pen.Width:=5;
     {Dessine une ellipse dont le coin supérieur gauche est à 0,0 et le coin
inférieur droit à 200,200}
     Form1.Canvas.Ellipse(0, 0, 200, 200);
end;
```

Le code suivant dessine le même cercle, au même emplacement sur le canevas de `TPrinter` et l'envoie à l'imprimante :

```
begin
     {début de la tâche d'impression }
     Printer.BeginDoc;
     { Définit une épaisseur de crayon de 5 pixels }
     Printer.Canvas.Pen.Width:=5;
    { Dessine une ellipse dont le coin supérieur gauche est à 0,0 et le coin
inférieur droit à 200,200}
     Printer.Canvas.Ellipse(0, 0, 200, 200);
     {fin et impression de la tâche d'impression }
     Printer.EndDoc;
end;
```

En ajoutant les lignes `BeginDoc` et `EndDoc`, et en transformant `Form1` en `Printer` pour pointer vers le canevas de l'imprimante plutôt que celui de la fiche, vous pouvez envoyer le même graphique à l'imprimante. Créons un petit programme comportant un bouton Ecran et un bouton Imprimante pour tester notre code.

Vous pouvez envoyer presque aussi facilement un graphique vers l'imprimante que vers l'écran.

Créons donc un programme qui permette à l'utilisateur de sélectionner un fichier graphique — bitmap (*.bmp) ou métafichier Windows (*.wmf) — dans une boîte de dialogue Ouvrir. Ce graphique sera affiché à l'écran puis envoyé à l'imprimante. C'est chose facile avec Delphi. Vous utilisez un composant Image pour stocker le graphique à l'écran et la méthode CopyRect pour déplacer l'image vers le canevas de l'imprimante, qui envoie ensuite l'image à l'imprimante. Dans ce programme, nous permettrons à l'utilisateur de sélectionner un diviseur qui modifiera la taille de la sortie papier. On ajoute également du code pour que le graphique soit imprimé au centre de la page.

Commençons par créer la fiche qui apparaît Figure 9.12, en utilisant les composants suivants :

- 2 boutons
- 1 composant `image`
- 1 composant `OpenDialog`
- 1 `Panel`
- 2 `Label`
- 1 composant `SpinEdit` pris dans l'onglet Exemples

Créez un nouveau projet. Appelez-le PRINTPIC.DPR et appelez l'unité PICPRINT.PAS. Référez-vous à la Figure 9.6 pour disposer les composants sur la fiche. Donnez à la fiche le titre `Impression d'image`. Modifiez les `caption` des boutons en Charger image et Imprimer. Ajoutez des filtres au composant `OpenDialog` pour qu'il contienne fichiers graphiques (.bmp et .wmf) et Tous (*.*). Dans le composant `SpinEdit`, définissez `MinValue` à 1 et `MaxValue` à 10. Donnez pour `caption` Diviseur de taille à `Label1` et Zone graphique à `Label2`. Définissez comme `True` la propriété `Image1.Stretch` pour que l'image soit redimensionnée de façon

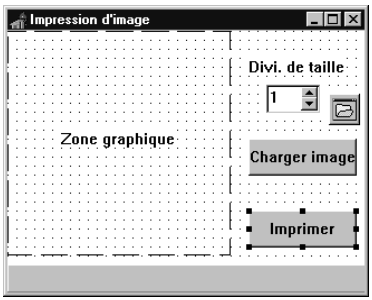


Figure 9.12
La fiche du programme d'impression d'image.

qu'elle entre dans la boîte d'image lors de son chargement. Assurez-vous également que la propriété `AutoSize` est définie comme `False`.

Note *Lorsque vous placez la boîte d'image sur la fiche, faites-en un carré de taille raisonnable et placez-la de sorte que ses propriétés `Top` et `Left` soient toutes deux à `0` dans l'Inspecteur d'objet. La boîte se mettra en position. Ce positionnement est important.*

Pour charger l'image, exécutez la méthode `LoadFromFile` pour la boîte d'image placée sur la fiche. Donnez comme paramètre pour la méthode `LoadFromFile` le nom de fichier provenant de `OpenDialog`. L'image est alors chargée puis affichée.

```
If OpenDialog1.Execute then
           Image1.Picture.LoadFromFile(OpenDialog1.FileName);
```

Pour imprimer l'image, vous devez savoir où se trouve le centre du canevas afin d'imprimer l'image au centre. Le code ci-après récupère les coordonnées x, y du centre en divisant la hauteur et la largeur de la page de l'imprimante par 2. Le résultat est stocké dans les variables `CenterX` et `CenterY`.

```
{On cherche le centre du canevas }
     PHeight:=Printer.PageHeight;
     PWidth:=Printer.PageWidth;
     CenterX:=PWidth Div 2;
     CenterY:=PHeight Div 2;
```

Vous calculez les coordonnées de positionnement en vous appuyant sur le diviseur entré par l'utilisateur. On commence par récupérer la valeur entrée dans la propriété `Value` du composant `SpinEdit`. Pour placer l'image, il faut connaître les coordonnées x,y des positions Haut, Gauche et Bas, Droite.

Les variables sont :

Variable	Position
`X1`	Gauche

Variable	Position
Y1	Haut
X2	Droite
Y2	Bas

Le code ci-après calcule la position du centre en utilisant le diviseur entré par l'utilisateur :

```
{On calcule la position du centre avec le diviseur de taille fourni par
l'utilisateur }
     SDiv:=SpinEdit1.Value;
     X1:=CenterX-(PWidth Div (SDiv*2));
     Y1:=CenterY-(PHeight Div (SDiv*2));
     X2:=CenterX+(PWidth Div (SDiv*2));
     Y2:=CenterY+(PHeight Div (SDiv*2));
```

Le diviseur est multiplié par deux. On divise ensuite la hauteur et la largeur par ce produit. Le résultat est soustrait ou ajouté aux valeurs CenterX et CenterY pour déterminer la position du centre. Si l'utilisateur sélectionne le diviseur par défaut (1), l'image est imprimée centrée sur la page et occupe la page toute entière. Si la page de l'imprimante mesure 2000 pixels par 2000 pixels, la position du centre est 1000,1000. En effet :

```
X1 = 1000 - (2000 Div (1*2))  soit     X1 = 1000 - 1000   soit
X1 = 0
```

Cela définit la position X1 (gauche) à 0, qui est la position la plus à gauche sur la page :

```
X2 = 1000 +  (2000 Div (1*2))  soit   X2 =   1000 + 1000    soit
X2 = 2000
```

Cela définit la position X2 (droite) à 2000, qui est la position la plus à droite sur la page :

Dans ce cas, l'image occupe toute la page. En changeant le diviseur, l'utilisateur peut réduire la taille de l'image imprimée. Vous utilisez l'opérateur Div pour les divisions car vous travaillez avec des entiers et vous ne souhaitez pas vous occuper des restes de division. Avec cette formule, l'image est toujours imprimée au centre de la page, quel que soit le diviseur entré par l'utilisateur.

Pour placer le graphique sur le canevas de l'imprimante, vous devez utiliser la méthode `CopyRect`, qui copie une zone rectangulaire d'un canevas vers un autre.

```
procedure CopyRect(Dest: TRect; Canvas: TCanvas; Source: TRect);
```

Les rectangles source et destination sont passés comme variables de type `TRect`. En l'occurrence, le canevas source est celui de `Form1`.

Vous placez la boîte d'image en 0,0 car la déclaration `CopyRect` prend l'image dans le canevas de la fiche, et pas directement dans la boîte d'image. Si vous placez la boîte d'image en 0,0 dans la zone client de la fiche, vous n'avez pas à prendre en compte de décalages dans vos cal-

culs. Si vous déplacez la boîte à image jusqu'à un nouvel emplacement sans calculer la valeur de décalage correspondante, vous risquez de perdre tout ou partie de l'image car celle-ci se trouvera alors en dehors de la zone rectangulaire qui est copiée.

Comme `CopyRect` prend deux variables de type TRect, vous devez déclarer deux variables de ce type et y placer les données. Pour déclarer les variables :

```
Var
       PrnRect, ImgRect : TRect;
```

`TRect` est défini ainsi comme enregistrement dans l'unité Windows :

```
TRect = record
  case Integer of
    0: (Left, Top, Right, Bottom: Integer);
    1: (TopLeft, BottomRight: TPoint);
end;
```

Vous lui fournissez quatre valeurs, puis vous utilisez la fonction `Rect` pour stocker les données dans les nouvelles variables. La fonction `Rect` prend les coordonnées X1, Y1, X2, Y2 (ou Haut, Gauche, Droite, Bas) comme paramètres et renvoie aux variables les données sous forme d'enregistrement `TRect`. Ainsi :

```
{On stocke la taille du rectangle de canevas d'impression dans PrnRect}
     PrnRect:=Rect(X1,Y1,X2,Y2);
     {on place la taille du Rect de la boîte d'image dans ImgRect}
     ImgRect:=Rect(0,0,Image1.Width,Image1.Height);
```

Une fois que vous avez affecté les valeurs `PrnRect` et `ImgRect`, utilisez les variables dans la déclaration de méthode `CopyRect` pour copier l'image dans le canevas de l'imprimante :

```
{On copie l'image de la boîte d'image au canevas de l'imprimante }
Printer.Canvas.CopyRect(PrnRect,Form1.Canvas,ImgRect );
```

A quelques détails près — vous devez aussi cacher le libellé qui marque la zone graphique lorsque le graphique est affiché et mettre à jour le volet en indiquant l'état du programme — votre programme d'impression d'image est maintenant fonctionnel. Le Listing 9.12 montre le code complet du programme d'impression de graphique. Entrez le code aux emplacements adéquats et testez le programme.

Listing 9.12 Impression d'image

```
unit Picprint;

interface
```

```
uses
 Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,Printers,StdCtrls, ExtCtrls, Spin;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    OpenDialog1: TOpenDialog;
    Image1: TImage;
    SpinEdit1: TSpinEdit;
    Panel1: TPanel;
    Label1: TLabel;
    Label2: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

Var
   PHeight,PWidth : Integer;


procedure TForm1.Button1Click(Sender: TObject);
Var
     PrnRect,ImgRect:TRect;
     CenterX,CenterY,X1,Y1,X2,Y2,SDiv : Integer;
begin
     Panel1.Caption:='Impression...';
     Printer.BeginDoc;
```

```
     {On cherche le centre du canevas }
     PHeight:=Printer.PageHeight;
     PWidth:=Printer.PageWidth;
     CenterX:=PWidth Div 2;
     CenterY:=PHeight Div 2;
    {On calcule la position du centre en tenant compte du diviseur de taille }
     SDiv:=SpinEdit1.Value;
     X1:=CenterX-(PWidth Div (SDiv*2));
     Y1:=CenterY-(PHeight Div (SDiv*2));
     X2:=CenterX+(PWidth Div (SDiv*2));
     Y2:=CenterY+(PHeight Div (SDiv*2));
     {On stocke la taille du rect de canevas d'impression dans PrnRect}
     PrnRect:=Rect(X1,Y1,X2,Y2);
     {On place le taille du rect de la boîte d'image dans ImgRect}
     ImgRect:=Rect(0,0,Image1.Width,Image1.Height);
     {On copie l'image de la boîte d'image vers le canevas de l'imprimante }
     Printer.Canvas.CopyRect(PrnRect,Form1.Canvas,ImgRect );
     Printer.EndDoc;
     Panel1.Caption:='Impression achevée!';
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
     {On charge un graphique dans la boîte d'iamge de la fiche }
     If OpenDialog1.Execute then
     begin
          Label2.Visible:=False;
          Image1.Picture.LoadFromFile(OpenDialog1.FileName);
     end;
end;

end.
```

Analyse *Le code du Listing 9.12 permet à l'utilisateur de sélectionner un fichier graphique en cliquant sur le bouton Charger image. L'image est affichée sur la fiche dans le composant Image. Puis, on peut imprimer l'image en cliquant sur le bouton Imprimer. Dans ce cas, l'image est copiée à partir du canevas de la fiche et est envoyée à l'imprimante.*

Vous devez pouvoir lancer le programme, sélectionner une image graphique et l'afficher à l'écran. Vous pouvez sélectionner un nombre de 1 à 10 comme diviseur de taille et imprimer l'image. Votre image est dimensionnée et centrée sur la page si tout s'est déroulé comme prévu.

Récapitulatif

Lors de cette journée, nous avons présenté les attributs de fichier et les méthodes pour les lire et les modifier. Vous avez appris à manipuler des fichiers texte en écrivant un éditeur de texte simple. Nous avons ensuite approfondi la question pour créer un programme simple de base de données permettant de stocker des adresses en utilisant des fichiers typés. Vous avez ensuite pu découvrir les fichiers non typés. Enfin, nous avons rapidement cité d'autres routines de gestion de fichiers et d'E/S. La question des manipulations de fichiers a été traitée de façon aussi complète que possible. Les contrôles de base de données et d'autres fonctionnalités de Delphi peuvent parfois rendre superflu ce type d'accès aux données. Cependant, les sujets abordés ici peuvent fournir matière à écrire des programmes de conversion, des utilitaires d'importation ou votre propre éditeur de texte spécialisé. Vous aurez forcément besoin un jour ou l'autre des notions que nous venons d'aborder. Quoi qu'il en soit, vous savez maintenant comment enregistrer, charger, lire et traiter des fichiers.

Vous avez également pu apprendre les techniques d'impression de base, consistant à traiter le port de l'imprimante comme un fichier et à utiliser des déclarations `writeln` pour envoyer du texte à l'imprimante. Vous avez pu écrire un programme lisant un fichier texte dont chaque ligne est tour à tour envoyée à l'imprimante.

Vous avez également appris à effectuer la même tâche en utilisant cette fois l'objet `TPrinter` qui vous permet de modifier les polices, la taille des polices, etc. Vous avez pu voir comment utiliser les boîtes de dialogue Impression pour permettre à l'utilisateur de sélectionner ses propres options d'impression. Vous avez vu comment dessiner sur le canevas de l'imprimante, comme si vous dessiniez sur l'écran avant d'envoyer l'image à l'imprimante.

Enfin, vous avez appris à charger une image graphique à l'écran et à l'envoyer à l'imprimante. Bien que cette journée n'ait fait qu'effleurer le sujet de l'impression, vous pouvez en appliquer les enseignements dans vos futures applications.

Vous devriez maintenant être en mesure de manipuler aussi bien les fichiers que l'impression. Vous pouvez dès lors créer des applications puissantes utilisant ces fonctionnalités essentielles.

Atelier

L'atelier vous donne trois façons de vérifier que vous avez correctement assimilé le contenu de cette journée. La section Questions - Réponses propose des questions fréquemment rencontrées, ainsi que leurs réponses, la section Questionnaire vous pose des questions, dont vous trouverez les réponses en consultant l'Annexe A, et les Exercices vous permettent de mettre en pratique ce que vous venez d'apprendre. Nous vous recommandons de lire et répondre à chacune de ces sections avant de passer à la suite ; elles constituent en effet un récapitulatif des points essentiels abordés lors de cette section.

Questions - Réponses

Q Y a-t-il quelque chose de particulier à faire pour utiliser les noms longs de Windows 95 et NT ?

R Non, les composants et fonctions de Delphi 3 comportent en standard toutes les fonctionnalités requises pour la gestion des noms longs. Ils sont à même de traiter indifféremment des noms courts ou des noms longs, à fin de compatibilité ascendante.

Q Que faut-il faire pour que les programmes reconnaissent le type de fichier auquel ils ont affaire ?

R Il n'y a rien à faire. Bien qu'il existe des standards sur la façon d'utiliser des extensions de fichier pour indiquer un type de fichier, ces standards ne sont pas toujours respectés. Vous devrez vous-même déterminer quel sera le formatage et la structure des fichiers à traiter et vous devrez faire de votre mieux pour vérifier la validité des fichiers et gérer les erreurs qui peuvent survenir. Plus généralement, vous devez espérer que l'utilisateur choisira le bon fichier.

Q J'ai essayé d'imprimer une icône en utilisant l'Impression d'image. J'ai réussi à la charger mais il n'a pas été possible de la dimensionner ou de la centrer correctement sur le papier ou sur l'écran. Pourquoi ?

R A la différence d'un bitmap ou d'un métafichier, une icône n'est pas un graphique redimensionnable dans la boîte d'image. L'icône s'affiche simplement dans le coin supérieur gauche de la boîte et s'imprime pareillement sur la page. Lorsque vous avez copié l'image rectangulaire de la boîte d'image vers le canevas de l'imprimante, vous avez pris la taille de la boîte d'image. L'icône n'utilise qu'une petite partie de cet espace. Le programme fonctionne correctement, le problème est que vous ne pouvez voir la boîte d'image qui contient l'icône. L'image change de taille et se déplace vers le centre lorsque vous spécifiez des diviseurs plus grands.

Q Comment savoir si ma tâche d'impression est bien en train de s'imprimer ?

R Utilisez la propriété `Printing` de l'objet `TPrinter`. Si `Printing` est défini comme True, la tâche d'impression a été appelée avec `BeginDoc` mais `EndDoc` n'a pas été appelé. Votre code pourra ressembler à ceci :

```
 If Printer.Printing then
{votre code ici}
```

Q Comment interrompre une tâche d'impression ?

R En utilisant la méthode Abort de l'objet TPrinter, vous pouvez ajouter du code à un bouton ou à votre programme pour interrompre une tâche d'impression. La ligne de code suivante provoque l'interruption de la tâche d'impression :

```
Printer.Abort;
```

Questionnaire

1. Quelle est la différence entre les fichiers typés et les fichiers non typés ?
2. Quelle procédure permet de déplacer le pointeur de fichier ?
3. Quels types de fichiers pouvez-vous manipuler si vous utilisez des fichiers non typés ?
4. Est-il nécessaire d'utiliser l'objet Printer pour envoyer du texte à l'imprimante ?
5. Pour utiliser les fonctions d'impression, que devez-vous faire figurer dans la partie interface de votre programme ?
6. Citez les deux méthodes importantes permettant d'imprimer avec l'objet Printer. (Indice : la tâche d'impression ne peut commencer ou s'achever sans elles).
7. Dessiner sur le canevas de l'imprimante n'est pas très différent de dessiner sur le canevas d'une fiche. Vrai ou faux ?
8. Quel code permet d'activer la boîte de dialogue Propriétés ?
9. Quelle méthode permet de copier une image du canevas de la fiche vers le canevas de l'imprimante ?
10. Comment imprimer un petit cercle ou une petite ellipse à l'aide de l'objet TPrinter ?

Exercices

1. Dans le programme de répertoire, ajoutez une fonctionnalité permettant de se déplacer jusqu'au premier ou au dernier enregistrement.
2. Modifiez le programme d'impression d'image pour permettre à l'utilisateur de décider s'il souhaite imprimer l'image au centre, en haut à gauche ou en bas à droite de la page.

10

Graphismes, multimédia et animation

Delphi propose de nombreuses fonctions permettant de créer facilement des applications graphiques. Lors de cette journée, nous vous présenterons la base de la création d'applications graphiques et vous montrerons comment utiliser des techniques graphiques et multimédias sophistiquées.

Les éléments graphiques

Les programmeurs doivent comprendre comment afficher des images dans des applications ou comment manipuler des points, des formes, des lignes et des couleurs. Windows 95 et Windows NT offrent des fonctionnalités puissantes permettant à des applications graphiques de haut niveau de tirer parti des ressources système le plus efficacement possible. Delphi vous offre une gamme étendue de composants et de méthodes cachant au développeur une grande partie des détails de l'implémentation système. C'est un plus pour celui qui découvre les graphismes car il peut se concentrer sur leur moyen de fonctionnement au lieu de perdre son temps à apprendre la manipulation d'appels complexes au système d'exploitation.

Coordonnées

Vous savez sans doute ce que sont des coordonnées. Tous les composants visuels ont une propriété `Top` (haut) et une propriété `Left` (gauche). Les valeurs stockées dans ces propriétés déterminent la position du composant sur la fiche. Autrement dit, le composant est placé aux coordonnées X, Y, où X est la propriété `Left` et Y la propriété `Top`. Les valeurs en X et Y (ou `Left` et `Top`) sont exprimées en pixels. Un pixel est la plus petite zone d'écran manipulable.

Propriété Canvas

La propriété `Canvas` est la zone de dessin sur une fiche et sur d'autres composants graphiques ; elle permet au code Delphi de manipuler la zone de dessin en cours d'exécution. L'une de ses principales caractéristiques est qu'elle est constituée de propriétés et de méthodes facilitant la manipulation de graphiques dans Delphi. Toutes les manipulations formelles et les comptabilités diverses sont cachées dans l'implémentation de l'objet `Canvas`.

La partie suivante présente les fonctions de base vous permettant d'effectuer des opérations graphiques dans Delphi à l'aide de l'objet `Canvas`.

Pixels

D'un point de vue conceptuel, toutes les opérations graphiques reviennent à définir la couleur des pixels de la surface de dessin. En Delphi, vous pouvez manipuler les pixels individuellement. Aux débuts de l'informatique, un pixel était soit activé, soit désactivé, il était donc noir ou blanc (ou vert ou ambre). Les pixels peuvent maintenant prendre une vaste gamme de couleurs. Leur couleur peut être spécifiée soit comme une couleur prédéfinie, telle que `clBlue`, soit comme un mélange arbitraire des trois couleurs rouge, vert et bleu.

Pour accéder aux pixels d'une fiche, vous devez utiliser la propriété `Canvas` de la fiche et la propriété `Pixels` du Canvas. La propriété `Pixels` est un tableau à deux dimensions correspondant aux couleurs du `Canvas.Pixels[10,20]` correspond à la couleur du pixel situé à 10 pixels à droite et à 20 pixels en bas de l'origine. Vous traitez le tableau pixel comme toute autre propriété : pour modifier la couleur d'un pixel, affectez-lui une nouvelle valeur ; pour déterminer sa couleur, lisez la valeur.

Utiliser des pixels

Dans l'exemple suivant, nous utilisons la propriété `Pixels` pour dessiner une courbe de sinus dans la fiche principale. Le seul composant de la fiche est un bouton qui dessine la courbe lorsqu'on clique dessus. Nous utilisons les paramètres `Width` (largeur) et `Height` (hauteur) de la fiche afin que la courbe prenne place sur 70 % de la hauteur de la fiche et sur l'ensemble de la longueur. Le code produisant cette courbe figure dans le Listing 10.1. La Figure 10.1 montre l'exécution de l'application.

Listing 10.1 : Dessiner une sinusoïde avec des pixels

```
unit unitSine;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs,
  StdCtrls;

type
  TForm1 = class(TForm)
    DrawSine: TButton;
    procedure DrawSineClick(Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  Form1: TForm1;

implementation
```

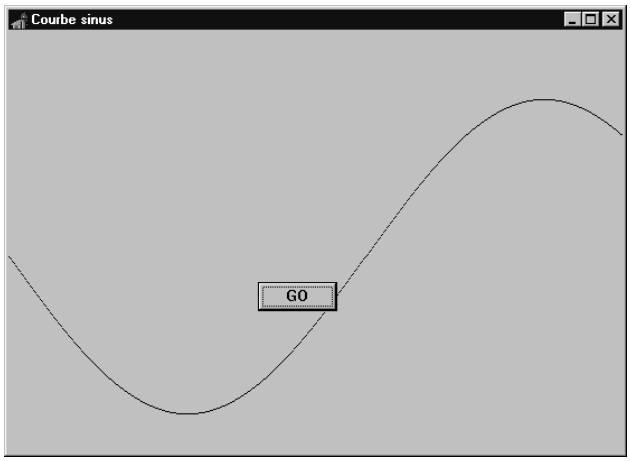
```
{$R *.DFM}

procedure TForm1.DrawSineClick(Sender: TObject);

var
  X  : real;
  Y  : real;
  PX :longint;
  PY :longint;
  HalfHeight : longint;
begin
{ On détermine la moitié inférieure de la fiche }
HalfHeight := Form1.Height div 2;
for PX:=0 to Form1.Width do
  BEGIN
{On met à l'échelle X en fonction de 2 PI pour décrire une période de sinus
complète }
    X  := PX * (2*PI/Form1.Width);
    Y  := sin(X);
    PY := trunc(0.7 * Y * HalfHeight) + HalfHeight;
    {On rend le pixel noir (0 d'intensité  RVB)}
    Canvas.Pixels[PX,PY] := 0;
  END;
end;
end.
```

Analyse *Dans cette application, on commence par déterminer la hauteur de la fiche afin d'échelonner correctement la sinusoïde. Le compteur PX va ensuite de 0 à la largeur de la fiche, et une valeur Y est calculée en chaque point de la sinusoïde. La coordonnée Y est multipliée par un facteur d'échelle afin que la courbe soit bien placée sur la fiche. Le point est dessiné en définissant la propriété* `Canvas.Pixel`.

Chaque canevas dispose d'un crayon imaginaire lui permettant de tracer des lignes et des formes. Pensez au crayon du canevas comme à un véritable crayon sur une feuille de papier. On peut déplacer ce crayon de deux façons. La première consiste à le déplacer en touchant le papier afin de laisser une marque. La deuxième consiste à le lever, ce qui ne laisse pas de marques. En outre, des attributs lui sont associés. Le crayon possède ainsi une couleur et une épaisseur de trait spécifiques. Pour le déplacer sans dessiner, il suffit d'utiliser la méthode `MoveTo`. La ligne de code ci-après déplace le crayon jusqu'aux coordonnées 23,56.

```
Form1.Canvas.MoveTo(23,56);
```

Figure 10.1
Le programme de sinusoïde utilisant la propriété Pixels. *La propriété* Pen.



Tracer des lignes

Pour tracer une ligne droite allant de la position actuelle du crayon jusqu'à une autre position, il suffit d'utiliser la méthode `LineTo`. `LineTo` n'a besoin que des coordonnées de la destination du crayon, et trace alors une ligne droite de la position actuelle jusqu'à la nouvelle.

La procédure simple du Listing 10.2 utilise la propriété `LineTo` pour dessiner un joli motif. La Figure 10.2 vous montre le programme.

Listing 10.2 : Un motif de lignes

```
procedure TForm1.DrawSineClick(Sender: TObject);
var
  X  : real;
  Y  : real;
  PX :longint;
  PY :longint;
  Offset : longint;
  HalfHeight : longint;
begin
{ On détermine les coordonnées de la moitié inférieure de la fiche }
HalfHeight := Form1.Height div 2;
For OffSet := -10 to 10 do
BEGIN
     PX := 0;
```

```
      While PX < Form1.Width  do
        BEGIN
            X := PX * (2*PI/Form1.Width);
            Y := sin(X);
            PY := trunc(0.7 * Y * HalfHeight)
             + HalfHeight + (Offset *10);
            IF (PX = 0) Then
              canvas.MoveTo(PX,PY);
            canvas.LineTO(PX,PY);
            PY := trunc(0.7 * Y * HalfHeight) +
               HalfHeight + ((Offset-1) *10);
            canvas.LineTO(PX,PY);
            PX := PX +15;
        END;
END;
end;
end.
```

Analyse

Ce programme est presque identique à celui du tracé de courbe sinus, à ceci près qu'ici nous créons des interstices entre les points, que nous connectons ensuite avec des lignes. Lors de la première itération (lorsque PX=0), on utilise la méthode `MoveTo` *pour aller au premier point. Il nous suffit ensuite d'utiliser* `MoveTo` *pour nous déplacer vers tous les points suivants.*

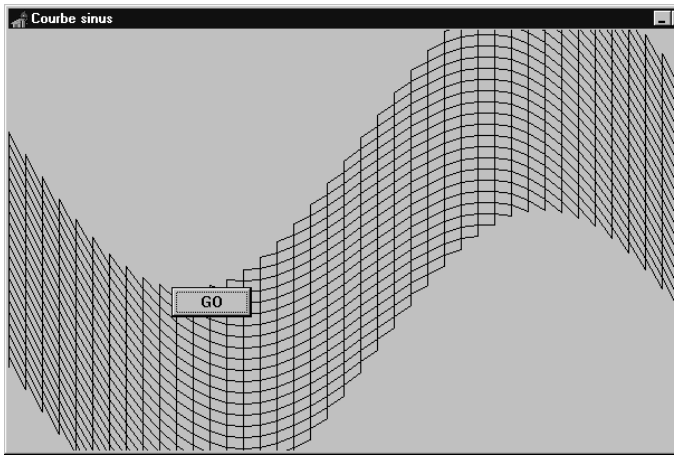
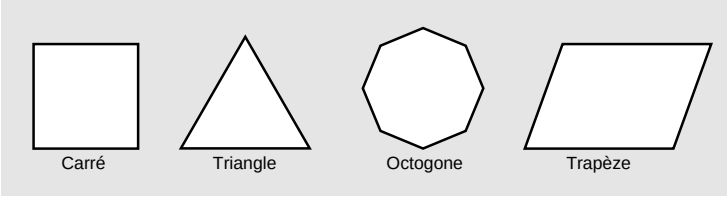


Figure 10.2
Motif dessiné à l'aide de LineTo *et de* MoveTo.

Dessiner des polygones

En plus du tracé de lignes droites, le canevas dispose de méthodes pour tracer des formes. Vous pouvez en tracer certaines, telles que des rectangles, en utilisant des méthodes spécifiques, mais vous pouvez en tracer d'autres en utilisant une série de points. Ces formes sont appelées polygones. Delphi dispose de plusieurs méthodes pour tracer des formes remplies. Voyons pour commencer les polygones contourés (nous verrons par la suite comment remplir des objets graphiques). On compte parmi les polygones usuels les triangles, les octogones ou les trapézoïdes. La Figure 10.3 vous montre quelques polygones usuels.

Figure 10.3
Quelques polygones courants.



Quelques polygones usuels

Pour tracer un polygone, vous passez à la fonction `PolyLine` une série de points qu'elle interconnecte au moyen de lignes.

Vous passez à la fonction `PolyLine` un tableau de point (c'est un concept qui doit vous sembler nouveau). Jusqu'ici, tout se faisait au moyen de coordonnées et vous passiez à la fonction `LineTo` des valeurs en X et Y.

Nouveau

Delphi comporte un type `TPoint`, qui encapsule les valeurs en X et Y dans un enregistrement unique appelé point. *La façon la plus simple de créer un point consiste à utiliser la fonction `Point`. `Point` prend une valeur X et une valeur Y et renvoie en enregistrement `TPoint`. Vous remarquerez que le premier et le dernier points ne sont pas forcément connectés, vous devez donc en spécifier un dernier identique au premier si vous souhaitez obtenir un polygone fermé. Ainsi, l'appel au PolyLine suivant trace un rectangle :*

```
Form1.Canvas.PolyLi-
ne([Point(10,10),Point(100,100),Point(50,75),Point(10,10)]);
```

Vous pouvez étendre ce principe pour créer une procédure traçant un polygone symétrique composé d'un nombre arbitraire de côtés entré par l'utilisateur. Si l'utilisateur entre `8`, la procédure trace un octogone ; un `6` donne un hexagone. Il suffit d'utiliser des principes simples de géométrie en plaçant les sommets sur un cercle. Le Listing 10.3 vous montre la source de ce programme.

Listing 10.3 : Un polygone à n côtés

```
procedure TForm1.DrawPolyClick(Sender: TObject);
Sides : integer;
```

```
  Count : integer;
  PolyArray : Array[0..15] of TPoint;

begin
    Sides := strtoint(NumSides.Text);
    If Sides > 15 then Sides := 15; /* Le tableau ne contient que 15 points*/
    For Count := 0 to Sides do
    BEGIN
{On utilise les points d'un cercle. On choisit des sommets comme points}
     PolyArray[Count] :=
       Point(TRUNC(SIN((2*PI)*COUNT/Sides)* 30)+(Form1.Width div 2),
             TRUNC(COS((2*PI)*COUNT/Sides)* 30)+(Form1.Height div 2));
    END;
{On connecte le dernier point au premier et on définit tous les }
     {points restants comme égaux au premier point }
    For Count := Sides+1 to 15 do
     PolyArray[Count] := PolyArray[0];
    {On dessine le polygone}
    Form1.Canvas.PolyLine(PolyArray);

end;
```

Analyse

Cette procédure calcule des points régulièrement espacés sur un cercle. Le nombre de points calculés dépend du nombre de côtés sélectionnés. Les points sont connectés en utilisant la méthode `PolyLines` *sur le canevas. Pour fermer le polygone, il suffit de définir les points inutilisés comme étant égaux au point de départ. La Figure 10.4 vous montre l'application en pleine exécution.*

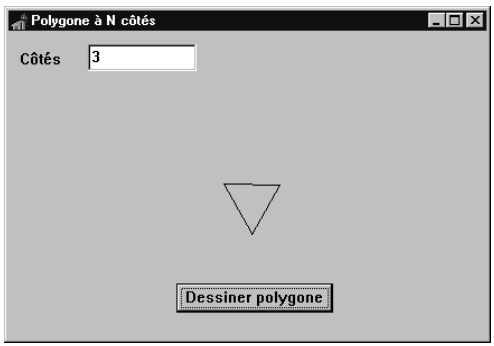


Figure 10.4
Un programme de tracé de polygone à n *côtés.*

Modifier les attributs de crayon

Toutes les formes tracées jusqu'ici utilisaient le crayon par défaut. Il est possible de changer la couleur, l'épaisseur de trait et le style du crayon. En Delphi, vous accédez au crayon par le biais du canevas, qui dispose d'une propriété `Pen`. Les principales propriétés par le crayon son `Color`, `Width`, `Style` et `Mode`.

Color

Vous pouvez définir la couleur du crayon en utilisant les mêmes méthodes que pour la couleur de la fiche. Ainsi, pour définir comme bleue la couleur du crayon, procédez ainsi :

```
Form1.Canvas.Pen.Color := clBlue;
```

Vous pouvez également utiliser la ligne suivante :

```
Form1.Canvas.Pen.Color := RGB(0,0,255);
```

Pour afficher toutes les nuances de gris, utilisez la procédure suivante :

```
procedure TForm1.DrawGreyClick(Sender: TObject);

var
  Count : Integer;
begin
  For Count := 0 to 255 do
  BEGIN
   Form1.Canvas.Pen.Color := RGB(Count,Count,Count);
   Form1.Canvas.MoveTo(Count,0);
   Form1.Canvas.LineTo(Count,100);
  end;
end;
```

Width et *Style*

La propriété `Width` définit la largeur du crayon en pixels. La propriété `Style` donne au crayon des tracés variés, tels que pointillés ou tirets. Les valeurs valides pour la propriété `Style` sont `psSolid`, `psDash`, `psDot`, `psDashDot`, `psDashDotDot`, `psClear` et `psInsideFrame`. Si vous souhaitez un crayon rouge, large de trois pixels et traçant en pointillé, exécutez ce qui suit :

```
code:
Form1.Canvas.Pen.Color := clRed;
Form1.Canvas.Pen.Width := 3;
Form1.Canvas.Pen.Style := psDot;
```

Mode

La propriété `Mode` du crayon lui permet d'interagir avec son environnement. Un mode `pmNot`, par exemple, fait tracer le crayon dans une couleur inverse de celle du fond (l'inverse de chaque bit par conséquent). Le crayon utilise le mode `pmCopy` par défaut, lui faisant utiliser la couleur courante. Vous pouvez déterminer la couleur du crayon en regardant la propriété `color`.

La propriété `Mode` vous permet de faire des animations simples. Vous pouvez créer une animation en redessinant une partie d'une image et en montrant les modifications au fur et à mesure. Le cerveau a alors l'illusion du mouvement. Pour des animations simples, vous pouvez utiliser les modes `pmXor` ou `pmNotXor` pour dessiner un objet, puis l'enlever sans modifier le fond. Rappelez-vous que chaque pixel stocke une couleur sous forme d'une séquence de bits. Modifier un pixel en effectuant un `Ou` exclusif (XOR) sur le pixel courant change la couleur. L'opération `XOR` prend deux opérandes considérés comme des ensembles de bits et renvoie `True` si l'un des opérandes est vrai (mais pas les deux). Procéder de même une nouvelle fois remet le pixel dans sa couleur initiale. Les étapes suivantes vous montrent ce qu'il se passe.

1. Le pixel de fond a une valeur de 0110011.
2. Vous effectuez un Ou exclusif de cette valeur avec 1111000.
3. Le nouveau pixel a une valeur de 1001011.

Lorsque vous souhaitez effacer le pixel et refaire apparaître le fond, il suffit de reprendre le même procédé :

1. Le pixel a actuellement pour valeur 1001011.
2. Vous effectuez un Ou exclusif de cette valeur avec 1111000.
3. Vous obtenez 0110011 comme résultat (la valeur initiale).

L'avantage de cette méthode est que vous n'avez pas besoin de stocker les informations concernant le fond, elles sont retrouvées automatiquement. L'inconvénient est que vous n'obtenez pas tout à fait l'image souhaitée car vous faites figurer les informations de fond dans la couleur du pixel. Dans la procédure du Listing 10.4, on utilise cette technique pour animer un triangle se déplaçant sur une fiche. Vous pouvez voir une boîte rouge sur la fiche, le triangle bleu passera sur la boîte sans la modifier.

Listing 10.4 : Une animation simple utilisant la propriété Mode

```
procedure TForm1.SimpleAnimateClick(Sender: TObject);
var
   Count : Integer;
   Pause : real ;
begin
{ On dessine une boîte }
Form1.Canvas.Pen.mode := pmCopy;
Form1.Canvas.Pen.Color := clRed;
```

```
Form1.Canvas.PolyLine([point(50,10),
                       point(100,10),
                       point(100,200),
                       point(50,200),
                       point(50,10)]);
{On définit le crayon }
Form1.Canvas.Pen.Color := clBlue;
Form1.Canvas.Pen.mode := pmNotXor;
For Count := 0 to (Form1.Width div 5) do
BEGIN
 {On dessine le triangle }
 Form1.Canvas.PolyLine([point(Count*5,100),
                       point(Count*5+10,100),
                       point(Count*5+5,110),
                       point(Count*5,100)]);
 Pause := Time;
 while (Time-Pause) < 1e-12 do; {nothing}
 {On efface le triangle }
 Form1.Canvas.PolyLine([point(Count*5,100),
                       point(Count*5+10,100),
                       point(Count*5+5,110),
                       point(Count*5,100)]);
end;  {du While}
end;
```

Analyse

Le programme commence le dessin d'une boîte rouge en définissant l'état du crayon à `pmCopy` et la couleur à `clRed`. Il déplace ensuite un triangle sur l'écran en utilisant le mode de crayon `pmNotXor`. Le mode `pmNotXor` est semblable à XOR car il préserve l'image de fond, mais il affiche la vraie couleur au premier plan. Il est nécessaire de dessiner deux fois le triangle à chacune de ses positions. La première fois, le triangle est dessiné, la seconde fois, il est effacé.

Objet Pinceau et remplissage

Au lieu de n'utiliser que les contours, vous pouvez remplir certains des objets proposés par Delphi. La propriété `Brush` détermine la façon dont un objet est rempli. Les trois propriétés principales affectant le pinceau (brush) sont `Color`, `Style` et `Bitmap`. On peut utiliser le pinceau de deux façons différentes : avec les propriétés `Color` et `Style` ou avec la propriété `Bitmap`.

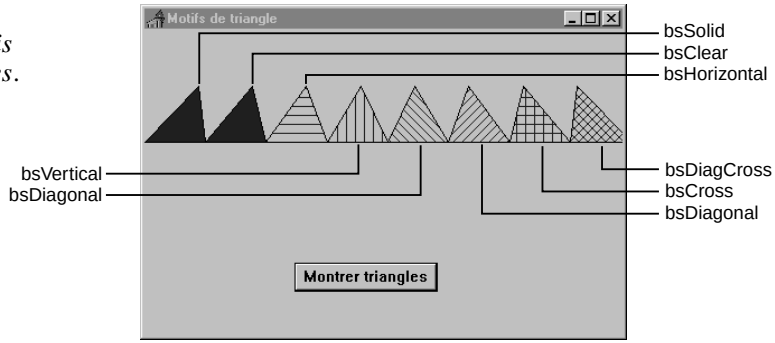
Lorsque vous utilisez les propriétés `Color` et `Style`, la couleur du remplissage dérive de la valeur de la propriété `Color`. La propriété `Style` définit le style du remplissage. De la même façon que vous utilisez la méthode `PolyLine` pour des objets contourés (non remplis), utilisez la méthode `Polygon` pour dessiner des polygones remplis.

L'exemple de programme du Listing 10.5 montre tous les styles disponibles sur huit triangles différents. La Figure 10.5 vous montre ces triangles.

Listing 10.5 : Tracé de triangles remplis dans différents styles

```
procedure Triangle(Iteration : Integer);
begin
Form1.Canvas.Brush.Color := clBlue;
Form1.Canvas.Polygon([Point(TRUNC((Iteration/9)*Form1.Width),50),
                      Point(TRUNC((Iteration/8)*Form1.Width),100),
                      Point(TRUNC(((Iteration-1)/8)*Form1.Width),100),
                      Point(TRUNC((Iteration/9)*Form1.Width),50)]);
end;

procedure TForm1.ShowTrianglesClick(Sender: TObject);
begin
  Form1.Canvas.Brush.Style := bsSolid;
  Triangle(1);
  Form1.Canvas.Brush.Style := bsClear;
  Triangle(2);
  Form1.Canvas.Brush.Style := bsHorizontal;
  Triangle(3);
  Form1.Canvas.Brush.Style := bsVertical;
  Triangle(4);
  Form1.Canvas.Brush.Style := bsFDiagonal;
  Triangle(5);
  Form1.Canvas.Brush.Style := bsBDiagonal;
  Triangle(6);
  Form1.Canvas.Brush.Style := bsCross;
  Triangle(7);
  Form1.Canvas.Brush.Style := bsDiagCross;
  Triangle(8);
end;
```

Figure 10.5
Des triangles remplis selon différents styles.



Analyse

Vous créez une procédure générique qui élabore des triangles dans une des huit régions de l'écran. La procédure `ShowTriangleClick` définit divers paramètres pour le pinceau et appelle la procédure de triangle en lui passant la position du triangle souhaité.

Au lieu d'utiliser les styles et les couleurs prédéfinis pour le pinceau, utilisez un bitmap qui définit le motif utilisé par le pinceau pour remplir des objets.

Un bitmap de pinceau est un bitmap de 8 pixels par 8 pixels définissant le motif utilisé pour remplir les objets.

Pour utiliser un bitmap sur un pinceau, vous devez commencer par créer un bitmap, l'affecter, puis le libérer lorsque vous avez terminé. La création et la manipulation des bitmap sont détaillées un peu plus loin.

Dessiner des rectangles remplis

De même que le type `TPoint` spécifie un ensemble de coordonnées en Delphi, un type `TRect` précise une partie rectangulaire dans une fiche ou dans une zone graphique. Vous spécifiez une région rectangulaire en donnant les coordonnées des coins supérieur gauche et inférieur droit. La fonction `Rect` permet de créer un type `TRect` à partir de coordonnées. La plupart des fonctions manipulant les régions rectangulaires utilisent des types `TRect` comme paramètres. Ainsi, vous utilisez la méthode `FillRect` pour dessiner un rectangle rempli. La ligne de code ci-après est un exemple d'utilisation de la méthode `FillRect`. Remarquez que vous devez utiliser la fonction `Rect` pour spécifier les coordonnées.

```
Form1.Canvas.FillRect(Rect(20,20,100,100));
```

En plus de la procédure `FillRect`, la procédure `Rectangle` dessine un rectangle en utilisant les attributs du pinceau courant pour le remplissage et ceux du crayon courant pour les contours. Cependant, cette procédure ne prend pas les mêmes paramètres que la précédente. Les quatre points sont paramètres, et l'on ne passe donc pas un type `TRect`. La ligne de code ci-après est un exemple d'utilisation de la procédure `Rectangle`.

```
Form1.Canvas.Rectangle(20,20,100,100);
```

Dessiner des cercles, des courbes et des ellipses

Tout ce que vous avez pu tracer jusqu'ici était constitué de points distincts ou de combinaisons de lignes droites. Le monde serait bien morne sans courbes ; Delphi propose différentes méthodes pour tracer des cercles, des ellipses, des arcs et des tranches. Un cercle est une ellipse dont le rayon est constant.

Pour dessiner une ellipse en Delphi, il suffit de fournir la région rectangulaire du canevas dans laquelle elle sera contenue. Pour tracer un cercle parfait, il suffit d'exécuter ce qui suit :

```
Form1.Canvas.Ellipse(100,100,200,200);
```

Pour dessiner une ellipse dont la largeur est supérieure à la hauteur, exécutez ce qui suit :

```
Form1.Canvas.Ellipse(100,100,300,200);
```

Pour ne dessiner qu'une portion d'ellipse, la procédure est un peu plus complexe. La méthode prend huit paramètres. Les quatre premiers sont nécessaires au tracé d'une ellipse complète. Les deux derniers (en fait quatre) sont les points indiquant le pourcentage de l'ellipse qui apparaîtra. Ils représentent les points d'arrivées des deux lignes partant de l'origine et définissant la portion de courbe à tracer. Ainsi, par exemple, la ligne ci-après trace le quart de cercle qui apparaît Figure 10.6.

```
Form1.Canvas.Pie(100,100,200,200,100,100,100,200);
```

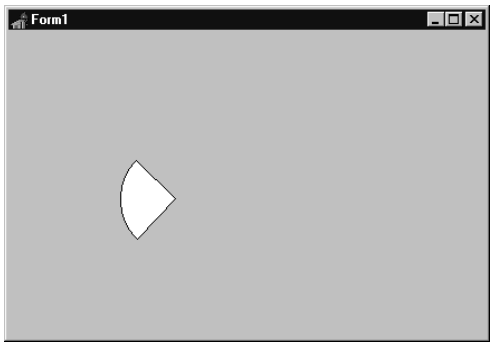


Figure 10.6
Un quart de cercle utilisant la méthode pie.

Un arc est en tous points semblable à une tranche, à cela près qu'il n'est pas rempli. Le code ci-après affiche un arc de même longueur que la tranche tracée ci-avant.

```
Form1.Canvas.Arc(100,100,200,200,100,100,100,200);
```

OnPaint : pour redessiner une fenêtre

Autrefois, un programme graphique utilisait tout l'écran et supposait que toute modification à l'écran était le fait des actions du programme lui-même. Dans un environnement Windows 95 ou Windows NT, de nombreuses applications peuvent s'exécuter simultanément sur un écran. Que se passe-t-il alors si une fenêtre est recouverte par une autre ou si elle est redimensionnée ? Le système peut garder une copie de l'écran en mémoire et effectuer en mémoire les modifications qui ne sont pas visibles. Cette méthode serait très coûteuse en ressources, tout particulièrement si de nombreuses applications s'exécutent. Au lieu de cela, le système d'exploitation prévient l'application que quelque chose a changé et c'est à l'application d'y remédier. Dès qu'une mise à jour est nécessaire, un événement `OnPaint` survient.

Delphi ne redessine que la partie du canevas qui a été affectée, ou invalidée. Lorsqu'une partie d'une fiche est invalide, Delphi appelle la procédure spécifiée dans le Gestionnaire d'événements `OnPaint`, afin de redessiner la partie invalidée de la fiche.

L'exemple du Listing 10.6 place dans le Gestionnaire d'événements `OnPaint` un code qui dessine un quart de cercle dans la fiche. Il fait également apparaître une boîte d'édition affichant le nombre de fois où la fiche a été repeinte.

Listing 10.6 : Démonstration du Gestionnaire d'événements OnPaint

```
unit unitOnPaint;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs,
  StdCtrls;

type
  TForm1 = class(TForm)
    NumRepaints: TEdit;
    procedure FormPaint(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { déclarations privées }
    NumPaints : integer;
  public
    { déclarations publiques }
  end;
```

```
var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormPaint(Sender: TObject);
begin
     NumPaints := NumPaints + 1;
     Form1.Canvas.Pie(100,100,200,200,100,100,100,200);
     NumRepaints.Text := IntToStr(NumPaints);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
    NumPaints := 0;
end;
end.
```

Analyse *Lorsque vous exécutez ce programme, vous pouvez voir le compteur d'itération augmenter chaque fois que le gestionnaire `OnPaint` est appelé. Cela est dû au fait que la variable `NumPaints` est incrémentée et affichée à chaque appel au gestionnaire `OnPaint`.*

Composant TPaintBox

Tous les dessins que vous avez effectués jusqu'ici l'ont été sur le canevas ou sur une fiche. Il est souvent utile de confiner un graphique à une région rectangulaire d'une fiche. Delphi propose un composant le permettant : `TPaintBox`. Faites l'exercice suivant :

1. Placez un composant `TPaintBox` sur une fiche.
2. Placez un composant `TButton` sur une fiche.
3. Ajoutez le code suivant dans l'événement `OnClick` du bouton :

   ```
   PaintBox1.canvas.Ellipse(0,0,2*PaintBox1.Width,2*PaintBox1.Hei-
   ght);
   ```

4. Compilez, puis exécutez l'application.

Que s'est-il passé ? Vous avez demandé une ellipse mais Delphi n'a tracé qu'un arc. En effet, l'ellipse est plus grande que le composant `TPaintBox`, qui est la seule zone dans laquelle Delphi peut dessiner. Le reste de l'ellipse a été tronqué. Imaginez la complexité d'une application qui

s'assurerait que rien n'est dessiné en dehors d'une région donnée. Le composant `TPaintBox` s'en charge pour vous. Les coordonnées dans une `TPaintBox` sont relatives à la `TPaintBox` elle-même, pas à la fiche. Cela signifie également que tant que le Gestionnaire d'événements `OnPaint` est responsable du tracé de la `TPaintBox`, vous pouvez déplacer l'image sur la fiche en modifiant les propriétés `Top` et `Left` de la `TPaintBox`. La `TPaintBox` utilise la propriété `Align` afin que l'image reste au sommet, sur la gauche, sur la droite ou au bas de la fiche. Cette propriété oblige également la `TPaintBox` à remplir la zone client de la fiche.

Composant TShape : moins de complexité

Que faire si vous souhaitez manipuler des formes simples, sans pour autant vous préoccuper de gestion d'événements pour contrer l'invalidation ? Existe-t-il des contrôles qui simplifient les actions ? Oui. Le composant `TShape` encapsule dans ses méthodes et propriétés la plupart des méthodes de dessin. Le composant `TShape` a des propriétés représentant son pinceau, son crayon et sa forme. Les formes que peut prendre le composant sont le cercle, l'ellipse, le rectangle, le rectangle arrondi, le carré et le carré arrondi. Le grand avantage de `TShape` est que tout le code nécessaire au tracé et au réaffichage de l'objet est caché.

Produire des images

Les méthodes et les composants graphiques conviennent parfaitement à la plupart des applications, mais il arrive qu'un développeur souhaite ajouter une image graphique prédessinée à son application. Cela serait assez dur à réaliser si l'on ne disposait que de composants graphiques. Prenons un exemple : les entreprises Bartlebooth, fabricants de puzzles, veulent placer la photo de leur fondateur sur tous les documents de l'entreprise. Obtenir ce résultat avec les méthodes graphiques serait un vrai cauchemar. Il vous suffit de prendre une photo de John Barnabooth et de la numériser en utilisant un scanner. Le programme de numérisation stocke l'image dans un format particulier que Delphi peut comprendre et se contente d'afficher. Les méthodes ne conviennent pas non plus si un artiste conçoit une image dans un programme de dessin et souhaite l'incorporer dans une application Delphi.

Delphi prend en charge de manière native quatre types d'images : les bitmap, les icônes, les métafichiers et les métafichiers avancés. Ces quatre types de fichiers stockent des images. La différence réside dans la façon dont les images sont stockées dans le fichier et dans les outils permettant de manipuler ces images et d'y accéder. Lorsque vous avez calculé la place prise en mémoire pour stocker une image de l'écran, vous avez dû multiplier la profondeur de couleur (en bits) par la résolution. Un bitmap est une sorte de photographie instantanée de l'écran et de toutes les informations qui lui sont associées. Un bitmap connaît la couleur de chaque pixel de l'image, mais ne sait pas ce que l'image représente. Ainsi, si vous prenez un bitmap d'un carré rouge sur fond bleu, les seuls informations présentes dans le bitmap sont que tous les pixels sont bleus, sauf les pixels qui appartiennent au carré dont les sommets sont (10,10) dans le coin supérieur droit et (100,100) dans le coin inférieur gauche, qui eux sont rouges.

Windows 3.1, Windows NT et Windows 95 ont édicté des formats de fichiers standard pour les bitmap. Les bitmap Windows sont des bitmap indépendants du matériel, ce qui signifie que les informations sont stockées de telle façon que n'importe quel ordinateur peut afficher l'image dans la résolution et avec le nombre de couleurs de sa définition. Cependant, cela ne veut pas dire que l'image a le même aspect sur n'importe quel ordinateur. Le résultat sera bien meilleur sur un écran acceptant une résolution de 1024 par 768 en couleurs 24 bits que sur un moniteur VGA standard. Le point à retenir est que les utilisateurs de ces deux ordinateurs pourront voir l'image. Ce standard permet au développeur de ne pas se soucier de la signification de chaque octet de ses fichiers bitmap.

Fort heureusement, les détails du format de fichier bitmap sont encapsulés dans le système d'exploitation et dans Delphi. Le moyen le plus facile d'afficher un bitmap en Delphi consiste à utiliser le composant `TImage`. Ce composant peut afficher différents types d'images graphiques. Il peut charger un bitmap provenant d'un fichier, et lui servir de conteneur dans l'application. Vous pouvez ainsi distribuer l'application sans devoir inclure un fichier bitmap distinct dans le logiciel.

Les icônes sont en fait de très petits bitmap. Elles appartiennent à une autre catégorie car elles servent généralement à figurer un raccourci vers une application ou une vue réduite d'objet. De manière interne, les icônes sont stockées comme les bitmap. Les métafichiers et les métafichiers avancés, en revanche, sont stockés de façon totalement différente. Un métafichier ne stocke pas des séries de bits décrivant l'image, mais des informations indiquant la façon dont l'image a été créée.

Les métafichiers stockent la séquence de commandes de tracé nécessaire à la re-création de l'image.

Pour afficher une image en utilisant le composant `TImage`, placez un `TImage` sur une fiche et double-cliquez sur la propriété `Picture`. Une boîte de dialogue apparaît alors, vous permettant d'afficher, de charger et d'enregistrer un bitmap dans le composant. Cliquez sur Charger et choisissez n'importe quel fichier .pico, .bmp, .emf ou .wmf valide. Le bitmap ou le métafichier que vous avez choisi est alors affiché dans le composant `TImage`.

Etirer et redimensionner des images

Par défaut, une image est affichée dans sa résolution d'origine, et vous n'en voyez que la partie qui est affichée dans le composant `TImage`. Deux propriétés importantes affectent la façon dont une image apparaît dans un `TImage`. La propriété `Autosize` fait que la taille du composant correspond aux dimensions de l'image. En définissant comme `True` la propriété `Stretch` (étirement) du composant, vous obligez l'image à prendre les dimensions du composant. Si vous définissez comme `False` les propriétés `Stretch` et `Autosize`, l'image est par défaut centrée dans le composant. Pour obliger l'image à s'afficher dans le coin supérieur gauche du composant, définissez comme `False` la propriété `Center`.

Charger en cours d'exécution une image provenant d'un fichier

Vous avez vu comment utiliser le composant `TImage` pour afficher un bitmap, un métafichier ou une icône en déclarant l'image au moment de la conception de votre application. Vous pouvez également charger un bitmap provenant d'un fichier en cours d'exécution à l'aide de la méthode `LoadFromFile`. La ligne qui suit charge le bitmap d'installation de Windows 95 dans votre composant d'image :

```
Image1.Picture.LoadFromFile('C:\WIN95\SETUP.BMP');
```

Remarquez que la méthode opère sur la propriété `Picture` du composant d'image et non sur le composant lui-même. L'image utilise la plupart de ses propriétés pour décrire la façon dont elle interagit avec l'application. La propriété `Picture` contient des informations sur l'image elle-même, vous devez donc charger l'image dans cette propriété.

Créer son propre bitmap

Nous vous avons montré précédemment comment dessiner sur le canevas d'une fiche et d'une Paintbox. Est-il possible de dessiner sur le canevas d'un bitmap ? La réponse est oui. En Delphi, un objet `TBitmap` possède un canevas que vous pouvez manipuler comme le canevas d'une `TPaintBox` ou d'une `TForm`. Lorsque vous dessinez sur un bitmap, il n'est pas nécessaire de vous soucier des événements `OnPaint` pour réafficher la scène en cas d'invalidation. Il vous suffit de recharger le bitmap qui est en mémoire. L'inconvénient du bitmap est qu'il nécessite plus de ressources système car il est stocké en mémoire. En Delphi, un bitmap seul est limité car il est difficile de l'afficher. Cela vient du fait que le bitmap n'est pas lui-même un composant et ne peut donc se "réparer" lui-même si quelque chose s'y inscrit. Cependant, si vous utilisez un bitmap en association avec un composant d'image qui l'affiche, ce composant image répond automatiquement à son propre événement `OnPaint` en redessinant le bitmap chaque fois que c'est nécessaire.

Créer entièrement un bitmap

Pour créer un nouveau bitmap, vous devez déclarer une variable de type `TBitmap` et utiliser la méthode `Create` comme constructeur pour allouer de l'espace au bitmap.

```
Var
   MyBitmap : TBitmap;

BEGIN
    MyBitmap := TBitmap.Create;
```

Pour l'instant, le bitmap a été créé mais il est encore vide. Il convient maintenant de définir ses dimensions. Utilisez pour cela les propriétés `Height` et `Width` :

```
MyBitmap.Height := 100;
   MyBitmap.Width  := 200;
```

Avant de dessiner le bitmap, ajoutez-lui quelques graphiques (ici, une ligne diagonale).

```
MyBitmap.Canvas.MoveTo(200,100);
  MyBitmap.Canvas.LineTo(0,0);
```

Pour afficher un bitmap, vous pouvez utiliser la méthode `Draw` qui en copie un sur un canevas. Toute autre manipulation du bitmap s'effectue en mémoire. Pour dessiner le bitmap sur Form1 aux coordonnées 100,100, utilisez la ligne suivante :

```
Form1.Canvas.Draw(100,100,MyBitmap);
```

Lorsque vous avez terminé avec le bitmap, vous devez libérer ses ressources système à l'aide de la méthode `Free` :

```
MyBitmap.Free;
```

Une méthode pour afficher le bitmap consiste à définir celui que vous avez créé comme l'image d'un composant d'image :

```
Image1.Picture.Graphic := MyBitmap;
```

L'un des avantages de cette méthode est que vous n'avez plus à vous soucier d'une éventuelle invalidation de l'image car le composant se charge de son réaffichage.

Enregistrer un bitmap dans un fichier

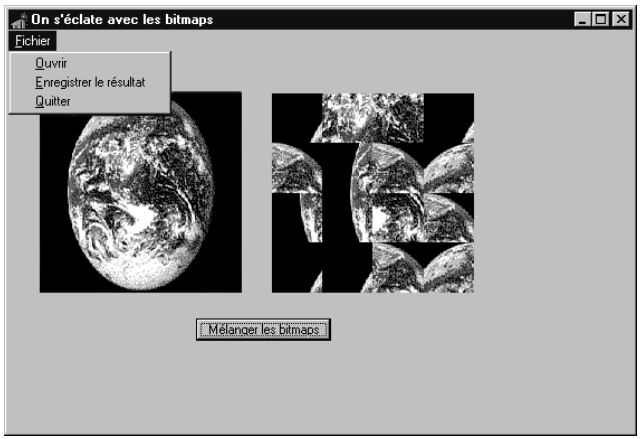
Vous pouvez non seulement charger ou manipuler des bitmap, mais aussi les enregistrer dans un fichier. Pour sauvegarder le bitmap, utilisez simplement la méthode `SaveToFile`.

```
My bitmap.SaveToFile ('C:\Perso\MyBitmap. BMP') ;
```

Un programme simple utilisant des images

Dans cet exemple, vous utilisez les fonctions graphiques de Delphi pour créer un programme qui manipule les bitmap. Un utilisateur charge un bitmap dans le programme, clique sur le bouton Mélanger bitmap, et crée ainsi un nouveau bitmap qui est construit en mélangeant aléatoirement des pièces du précédent. La Figure 10.7 montre le programme en action.

Figure 10.7
Le programme de mélange d'images.



Le Listing 10.7 montre le source de l'unité mixup.

Listing 10.7 : Le programme Mixup

```
unit unitMixup;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs,
  StdCtrls, Menus, ExtCtrls;

type
  TForm1 = class(TForm)
    Orig: TImage;
    NewImage: TImage;
    OpenDialog1: TOpenDialog;
    SaveDialog1: TSaveDialog;
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Save1: TMenuItem;
    Open1: TMenuItem;
    Exit1: TMenuItem;
    Mixup: TButton;
    procedure Exit1Click(Sender: TObject);
    procedure Open1Click(Sender: TObject);
```

```
    procedure MixupClick(Sender: TObject);
    procedure Save1Click(Sender: TObject);
  private
    { déclarations priées  }
    NewBitmap : TBitmap;  {Bitmap to create}
    ImageSet  : Boolean;
  public
    { déclarations publiques }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Exit1Click(Sender: TObject);
begin
   Application.Terminate;
end;

procedure TForm1.Open1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
   Begin
      Orig.Picture.LoadFromFile(OpenDialog1.FileName);
   End;
end;

procedure TForm1.MixupClick(Sender: TObject);
{ Divise le bitmap initial en 16 morceaux.  Crée un}
{ nouveau bitmap en sélectionnant de façon aléatoire un des 16}
{ morceaux et ce pour chaque morceau du nouveau bitmap }

Var
  NewBitmap     : TBitmap;  {Bitmap à créer }
  X,Y,NewX,NewY : integer;  {utiliser pour indexer les pièces }
  ChunkX,ChunkY : integer;  {dimensions X et Y de chaque pièce }

begin
```

```
  {libère Bitmap pour en charger un nouveau }
  if ImageSet then
     NewBitmap.Free;

  {On alloue des ressources pour un nouveau bitmap }
  NewBitmap := TBitmap.Create;

{les dimensions sont celles de l'original }
  NewBitmap.Height := Orig.Picture.Bitmap.Height;
  NewBitmap.Width := Orig.Picture.Bitmap.Width;
  NewImage.Stretch := true;

{On calcule les dimensions de chaque section }
  ChunkX := NewBitmap.Width div 4;
  ChunkY := NewBitmap.Height div 4;

{On construit le nouveau bitmap }
  For X := 0 to 3 do
   For Y := 0 to 3 do
    BEGIN
      NewX := random(3);
      NewY := random(3);
      NewBitmap.canvas.CopyRect(
         Rect(X*ChunkX,Y*ChunkY,(X+1)*ChunkX,(Y+1)*ChunkY),
         Orig.Picture.Bitmap.Canvas,
         Rect(NewX*ChunkX,NewY*ChunkY,
             (NewX+1)*ChunkX,(NewY+1)*ChunkY));
    END; {For Y}

  {on affiche le nouveau bitmap }
  NewImage.Picture.Graphic := NewBitmap;
end;

procedure TForm1.Save1Click(Sender: TObject);
{On utilise la boîte de dialogue Enregistrer usuelle pour enregistrer le
bitmap }
begin
  if SaveDialog1.Execute then
    Begin
      NewImage.Picture.SaveToFile(SaveDialog1.FileName);
```

```
    End; {if}
end; {Procedure}

procedure TForm1.FormCreate(Sender: TObject);
begin
    ImageSet := False;
end;

end.
```

Analyse

Examinez le fonctionnement de ce programme. On commence par utiliser les composants suivants :

Deux composants d'image permettant d'afficher l'image initiale et l'image mélangée.

Un composant `OpenDialog` *pour accéder à la boîte de dialogue usuelle Ouvrir.*

Un composant `SaveDialog` *pour accéder à la boîte de dialogue usuelle Enregistrer.*

Un menu et un bouton standard pour déclencher des événements.

La Figure 10.8 montre la disposition des divers composants.

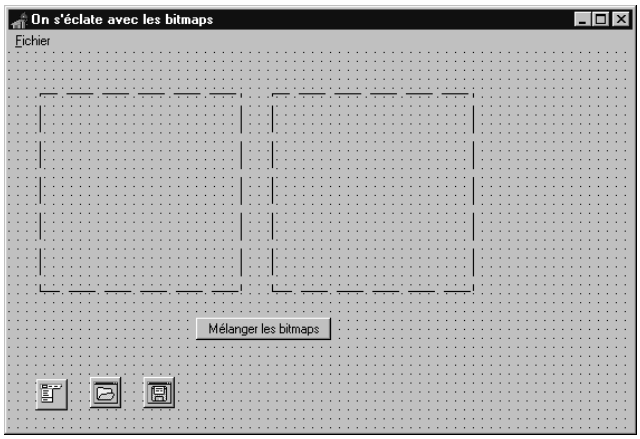


Figure 10.8
Disposition des composants de Mixup.

Lorsque vous sélectionnez l'option Fichier|Ouvrir dans le menu, le programme affiche la boîte de dialogue Ouvrir usuelle. En choisissant OK, le programme charge le fichier de bitmap que vous avez sélectionné dans le composant d'image `Orig`, à l'aide du code suivant :

```
Orig.Picture.LoadFromFile(OpenDialog1.FileName);
```

Ensuite, vous cliquez sur le bouton Mélanger les bitmap cela invoque la procédure `MixupBitmapClick` qui mélange le bitmap. Les étapes qui permettent de construire un nouveau bitmap et de prendre des morceaux de façon aléatoire dans le bitmap initial sont les suivantes :

1. Vous libérez d'éventuels bitmap existants puis construisez un nouveau bitmap en exécutant la méthode `Create` sur un objet bitmap de cette façon :

```
NewBitmap := TBitmap.Create;
```

2. Définissez les dimensions du bitmap pour qu'elles soient égales à celles de l'image initiale. Vous pouvez remarquer qu'on utilise `Picture.Bitmap` au lieu de `Picture.Graphic` pour obtenir ces dimensions.

```
NewBitmap.Height := Orig.Picture.Bitmap.Height;
NewBitmap.Width := Orig.Picture.Bitmap.Width;
```

3. Lorsque le bitmap initial est divisé en 16 morceaux et que le nouveau bitmap est assemblé, vous devez connaître la taille de chaque morceau. On calcule une valeur entière pour quatre morceaux dans chaque direction. La perte éventuelle de trois bits par partie est acceptable (ainsi, si le bitmap est large de 43 pixels, `ChunkX` ne fera que 10 pixels).

```
ChunkX := NewBitmap.Width div 4;
ChunkY := NewBitmap.Height div 4;
```

4. Chaque secteur est ainsi parcouru et vous sélectionnez une pièce aléatoire que vous placez dans le nouveau bitmap. Pour cela, construisez une boucle imbriquée qui choisit les différents morceaux du bitmap à créer. Vous sélectionnez un morceau aléatoire dans le bitmap initial en utilisant la fonction `random` qui donne des coordonnées X et Y aléatoires. Vous pouvez dessiner un bitmap en utilisant une fonction de canevas quelconque. Une de ces fonctions est `CopyRect`, elle copie une région rectangulaire d'un canevas sur une région rectangulaire d'un autre canevas.

```
For X := 0 to 3 do
   For Y := 0 to 3 do
    BEGIN
      NewX := random(3);
      NewY := random(3);
      NewBitmap.canvas.CopyRect(
         Rect(X*ChunkX,Y*ChunkY,(X+1)*ChunkX,(Y+1)*ChunkY),
         Orig.Picture.Bitmap.Canvas,
         Rect(NewX*ChunkX,NewY*ChunkY,
             (NewX+1)*ChunkX,(NewY+1)*ChunkY));
    END; {For Y}
```

5. Lorsque le programme crée le nouveau bitmap, il l'affiche en définissant la propriété `Picture.Graphic` de la nouvelle image comme étant égale au bitmap, à l'aide de la ligne de code suivante :

```
NewImage.Picture.Graphic := NewBitmap;
```

Pour enregistrer l'image, le programme utilise la boîte de dialogue Enregistrer usuelle lorsque vous sélectionnez Fichier|Enregistrer dans le menu. Si vous donnez un fichier valide, le programme utilise la méthode `SaveToFile` sur le composant `NewImage`, comme le montre le code suivant :

```
NewImage.Picture.SaveToFile(SaveDialog1.FileName);
```

Multimédia et animation

Qu'est-ce que le multimédia ? Pour simplifier, les ordinateurs actuels peuvent exprimer l'information par un autre médium que de simples images fixes sur un écran. Nous pouvons visualiser de véritables vidéos et écouter des sons. L'utilisateur est toujours étonné de pouvoir entendre un discours ou visionner un clip vidéo.

Delphi fournit des composants et des appels API qui rendent la programmation multimédia facile.

Pour commencer : utiliser des sons dans les applications

Il existe bien des façons de lire un son dans une application. Dans cette partie, nous verrons un moyen très simple d'utilisation du multimédia dans Delphi, avec l'appel API `PlaySound()`. L'appel API `PlaySound()` peut lire des sons provenant de différentes sources en employant diverses options. Vous pouvez utiliser cet appel pour lire un fichier WAV ou pour lire un des sons système par défaut. Vous disposez aussi de nombreuses options affectant la manière dont le son interagit avec votre application. Ainsi, vous pouvez lire un son et suspendre l'exécution du programme jusqu'à son achèvement, ou vous pouvez lire un son et continuer à exécuter d'autres commandes Delphi alors que sa lecture se poursuit.

Commençons avec un exemple simple. Créons une nouvelle application et ajoutons un unique bouton sur la fiche principale. Dans l'unité de la fiche principale, deux étapes sont nécessaires pour lire un son. Tout d'abord, l'unité `mmsystem` doit être ajoutée dans une clause `uses` de la partie implémentation de l'unité. Ensuite, l'appel API `PlaySound` doit être ajouté au Gestionnaire d'événements `OnClick` du composant visuel `TButton`. Pour notre premier exemple, nous allons nous contenter de lire un des fichiers WAV fournis avec Windows 95. Vous spécifiez

également que le son doit être joué de façon synchrone, ce qui signifie que l'application est suspendue jusqu'à la fin de la lecture du fichier son. Le code de cette unité figure dans le Listing 10.8.

Listing 10.8 : Un son dans une application Delphi avec l'appel API PlaySound()

```
unit mult1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  Form1: TForm1;

implementation
{Remarque : il faut inclure mmsystem pour que le programme fonctionne}
uses mmsystem;

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
begin
     {Remplacez C:\WIN95 par l'emplacement de votre répertoire Windows 95 }
     PlaySound('C:\WIN95\MEDIA\The Microsoft Sound', 0, SND_SYNC);
end;
```

Dans cet exemple, on se contente de lire le fichier son se trouvant dans le sous-répertoire C:\WIN95\MEDIA appelé "The Microsoft Sound". Le son est joué de façon synchrone, c'est-à-dire que le programme est suspendu jusqu'à la fin de la lecture.

Les trois paramètres pris par l'appel API `PlaySound` sont les suivants :

- Le paramètre 1 est un paramètre de variante qui peut être un nom de fichier, une adresse mémoire ou un événement système.
- Le paramètre 2 est utilisé dans des occasions spéciales, lorsque le son est stocké dans un fichier de ressources. Cette section ne traitera pas d'un tel cas, nous supposerons donc que le paramètre 2 est à 0.
- Le paramètre 3 spécifie toutes les options, notamment la façon de jouer le son et le type de paramètre que représente le paramètre 1. N'oubliez pas que l'API Win32 n'est pas orientée objet, il ne peut donc y avoir de formes multiples d'un appel qui détecterait automatiquement le type de données contenues dans le premier paramètre. Les constantes utilisées pour le paramètre 3 sont définies dans l'unité mmsystem. Des opérateurs logiques ou des opérations sur les bits peuvent être utilisés pour combiner plusieurs options. Ainsi, l'appel ci-après joue "The Microsoft Sound" de façon asynchrone (permettant la poursuite de l'exécution pendant la lecture du son) et continue à jouer le son jusqu'à ce qu'un autre appel à `PlaySound` soit effectué.

```
PlaySound('C:\WIN95\MEDIA\The Microsoft Sound', 0, SND_ASYNC or
SND_LOOP);
```

Les événements système sont des sons prédéfinis qui surviennent lorsqu'un programme ou le système d'exploitation désirent faire passer un message clair à l'utilisateur. Par exemple, pour jouer le son qui retentit lors du démarrage du système, vous pouvez faire :

```
PlaySound('SystemStart', 0, SND_ASYNC OR SND_NODEFAULT);
```

Si vous souhaitez jouer un son régulièrement et ne voulez pas accéder chaque fois au disque, il est possible de jouer un son stocké en mémoire. Ce n'est généralement pas nécessaire car le système d'exploitation se charge très bien de gérer un cache disque. L'avantage de cette méthode est que votre utilisateur n'a pas accès au fichier (ainsi, vous empêchez quiconque de repiquer votre son).

Différents types de fichiers multimédias

L'API `PlaySound` utilisait un fichier particulier appelé fichier wav (de *wave*, onde en anglais). Vous saviez qu'un fichier wav pouvait stocker un son, mais comment fait-il exactement ? En stockant une représentation numérique de la fréquence et du volume sur la longueur du clip sonore. Le fichier wav ne sait pas ce que représente le son, et par conséquent il doit stocker de nombreuses informations pour contenir un clip sonore.

Un autre type de fichier multimédia est le fichier MIDI (Musical Instrument Digital Interface ou Interface musicale d'instrument musical). Un fichier MIDI stocke le son en enregistrant des données indiquant quels instruments jouent quelles notes sur quelle durée. C'est l'équivalent informatique d'une partition de chef d'orchestre. L'un des grands avantages des fichiers MIDI est leur taille très réduite. Pour employer une analogie, le fichier wav est au fichier MIDI ce

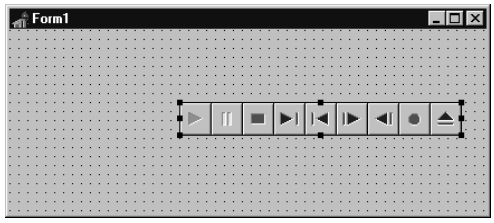
qu'un fichier bitmap est à un métafichier. Dans les deux cas, un format de fichier comprend les données qu'il représente et l'autre se contente de contenir les données brutes qui sont ensuite transmises à un périphérique de sortie.

Les fichiers Wav et MIDI ne peuvent stocker que des sons. Qu'en est-il de la vidéo ou de l'animation ? Il existe de nombreux formats de stockage de la vidéo, AVI et MPEG par exemple. La plupart des fichiers vidéo contiennent également une piste réservée au son, afin que le son et l'image soit synchronisés. Maintenant que vous savez quels sont les formats de fichier utilisés, vous devez savoir les utiliser dans Delphi. C'est là qu'intervient le composant visuel Media Player (lecteur multimédia), présenté dans la partie suivante.

Composant visuel Lecteur multimédia

La MCI (*Media Control Interface*, Interface de contrôle de support) est une interface de commande de haut niveau permettant de contrôler les fichiers multimédias, et qui est intégrée aux systèmes d'exploitation Windows 95 et Windows NT. Delphi fournit un composant visuel qui encapsule l'interface de commande dans ce qui ressemble à un magnétoscope ou à un magnétophone (voir Figure 10.9).

Figure 10.9
*Le composant visuel Lecteur multimédia (*MediaPlayer*).*



Le composant `Media Player` peut être utilisé de deux façons. Il contient une interface utilisateur pouvant être activée, permettant ainsi à l'utilisateur de manipuler des fonctions multimédias à l'aide de boutons tels que "Lecture", "Stop", "Enregistrer" et "Arrière". La seconde façon consiste à cacher le composant et à manipuler les fichiers multimédias en exécutant les méthodes du composant à partir d'un bloc de code.

Commençons par créer une application simple mais puissante permettant à l'utilisateur de charger un fichier WAW, MIDI ou AVI, puis de le lire et de le manipuler à l'aide de l'interface du composant `Media Player`. L'application, assez simple, nécessite trois composants : un composant `TMainMenu`, un composant `TOpenDialog` et un composant `TMediaPlayer`. Ajoutez la possibilité de sélectionner Fichier|Ouvrir dans le composant de menu. Les deux autres composants doivent être placés sur la fiche comme le montre la Figure 10.10.

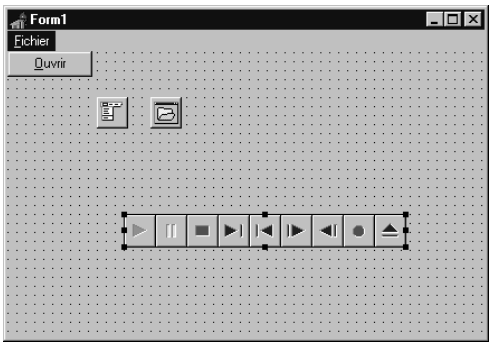


Figure 10.10
Disposition des composants pour le lecteur multimédia simple.

Ajoutez le code ci-après à l'événement Fichier/Ouvrir du menu :

```
procedure TForm1.Open1Click(Sender: TObject);
begin
{On définit le nom du fichier }
   MediaPlayer1.FileName := OpenDialog1.FileName;
   {On ouvre le fichier à l'aide de la méthode open }
   MediaPlayer1.Open;
 end;
```

Vous disposez maintenant d'une application pleinement fonctionnelle pouvant lire des fichiers audio et vidéo au moyen d'une interface des plus conviviales. La Figure 10.11 montre cette application lisant un fichier AVI standard.

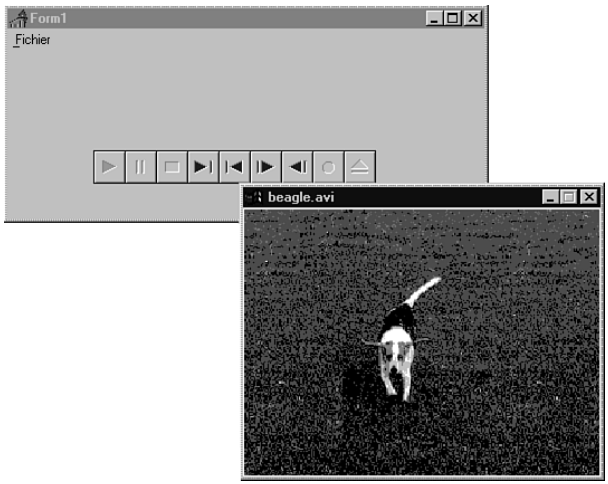


Figure 10.11
Le lecteur multimédia simple lisant un fichier AVI.

Il est possible de modifier la façon dont le media player interagit avec les fichiers multimédias en modifiant les propriétés du composant. Lorsque un fichier AVI est lu, le composant affiche la vidéo dans sa propre fenêtre. Ce comportement peut être changé en modifiant la propriété `Display` du media player. Cette propriété `Display` indique au lecteur multimédia où afficher le fichier vidéo qu'il lit. La valeur peut être n'importe quelle fiche ou composant dérivés d'un `TWinControl`. Il existe également une propriété `DisplayRect` permettant de spécifier la région de la nouvelle fenêtre dans laquelle sera affichée le clip vidéo.

La propriété `DisplayRect` peut prêter à confusion dans sa façon d'accepter ses paramètres. On affecte le type `TRect` à la propriété `DisplayRect` et l'on pourrait penser que ce type représente la région dans laquelle l'image sera affichée. Ce n'est pas le cas, en fait le paramètre haut-gauche indique bien où placer l'image, mais le paramètre bas-droite spécifie la largeur et la hauteur. Ainsi,

```
MediaPlayer1.Display := Form1;
MediaPlayer1.DisplayRect := RECT(10,10,200,200);
```

indique que vous souhaitez afficher la vidéo sur la Form1 entre les coordonnées 10,10 et 200,200 (et non pas 10,10 à 200,200).

En utilisant ces deux propriétés, nous allons créer notre deuxième application d'exemple, telle qu'elle figure dans le Listing 10.9. Cette application permet à l'utilisateur de charger un fichier vidéo et de le jouer comme auparavant. Cependant, l'image sera maintenant affichée dans une partie de la fiche principale et non plus dans sa propre fenêtre. De plus, vous ajoutez un bouton spécial permettant à l'utilisateur de capturer une image particulière. L'image capturée est conservée même si la lecture du clip vidéo se poursuit. La Figure 10.12 montre un fichier AVI en train d'être lu dans le cadre de gauche tandis qu'une image a été capturée dans le cadre de droite.

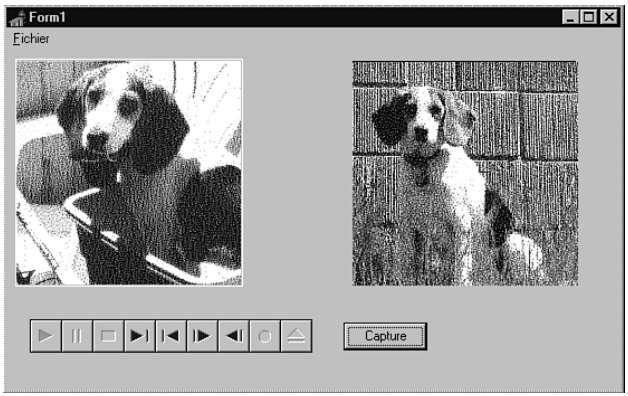


Figure 10.12
L'application de capture d'image.

Listing 10.9 : L'application de capture d'image

```
unit unitCapture;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs,
  ExtCtrls, StdCtrls, Menus, MPlayer;

type
  TForm1 = class(TForm)
    MediaPlayer1: TMediaPlayer;
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Open1: TMenuItem;
    Capture: TButton;
    PaintBox1: TPaintBox;
    Exit1: TMenuItem;
    OpenDialog1: TOpenDialog;
    procedure Open1Click(Sender: TObject);
    procedure CaptureClick(Sender: TObject);
    procedure FormPaint(Sender: TObject);
    procedure Exit1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure PaintBox1Paint(Sender: TObject);
  private
    { Déclarations privées }
    ImgBitmap : TBitmap;
  public
    { Déclarations publiques }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
```

```
procedure TForm1.Open1Click(Sender: TObject);
{on ouvre le fichier et on définit l'affichage sur form1}
{on définit également la région sur laquelle afficher (10,10,210,210)}
begin
 {On utilise le composant OpenDialog pour trouver un fichier vidéo }
 if OpenDialog1.Execute then
 begin
   MediaPlayer1.FileName := OpenDialog1.FileName;
   MediaPlayer1.Open;
   MediaPlayer1.Display := Form1;
   MediaPlayer1.DisplayRect := RECT(10,10,200,200);
 end; {if}
end; {procedure}

procedure TForm1.CaptureClick(Sender: TObject);
{lorsqu'on appuie sur le bouton de capture, on utilise la méthode CopyRect }
{pour copier l'image dans un bitmap en mémoire }
begin
    ImgBitmap.Canvas.CopyRect(Rect(0,0,200,200),
                              Form1.Canvas,Rect(10,10,210,210));
    PaintBox1.Invalidate;
end; {procedure}

procedure TForm1.FormPaint(Sender: TObject);
{Lorsque la fiche est invalidée, on redessine le rectangle en arièrre-plan }
begin
  Canvas.FrameRect(Rect(8,8,212,212));
end; {procedure}

procedure TForm1.Exit1Click(Sender: TObject);
begin
    Application.terminate;
end;

procedure TForm1.FormCreate(Sender: TObject);
{lorsque la fiche est créée on alloue des ressources pour le }
{bitmap et on définit la taille initiale. On efface également le bitmap.}
begin
  ImgBitmap := Tbitmap.create;
  ImgBitmap.Height := 200;
```

```
  ImgBitmap.Width  := 200;
  ImgBitmap.Canvas.Rectangle(0,0,200,200);
end; {procedure}

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
{A la fin, on nettoie en libérant le bitmap }
begin
  ImgBitmap.Free;
end; {procedure}

procedure TForm1.PaintBox1Paint(Sender: TObject);
{lorsque la paintbox qui affiche l'image capturée est invalidée,}
{on copie le bitmap présent en mémoire dans la paintbox.
{On évite ainsi de perdre l'image }
begin
   PaintBox1.Canvas.CopyRect(Rect(0,0,200,200),
                             ImgBitmap.Canvas,Rect(0,0,200,200));
end; {procedure}

end.
```

Analyse

Pour charger le fichier, vous définissez une option Ouvrir dans le menu principal. Le Gestionnaire d'événements, appelé lorsque l'utilisateur ouvre un nouveau fichier, charge le fichier, détermine l'endroit où sera affiché le clip vidéo et active le composant media player.

Pour déterminer le fichier à ouvrir, utilisez le composant `OpenDialog`. Lorsque la méthode `execute` est appliquée à ce composant, la boîte de dialogue Ouvrir standard apparaît et une valeur booléenne est renvoyée, indiquant si l'utilisateur a choisi "OK" ou "Annuler". Si l'utilisateur sélectionne un fichier valide, on affecte à la propriété `FileName` du media player la propriété `FileName` du composant `OpenDialog` :

```
MediaPlayer1.FileName := OpenDialog1.FileName;
```

Ensuite, on ouvre le fichier et on active le media player en appelant la méthode `Open` :

```
MediaPlayer1.Open;
```

La dernière étape de la préparation du clip vidéo consiste à spécifier que vous souhaitez voir l'image apparaître sur la Form1, et qu'elle reste circonscrite dans un rectangle bien défini. Cette précaution est nécessaire afin que vous sachiez où capturer l'image :

```
MediaPlayer1.Display := Form1;
MediaPlayer1.DisplayRect := RECT(10,10,200,200);
```

Le programme utilise l'interface fournie avec le media player pour permettre à l'utilisateur de manipuler le fichier multimédia chargé. Il est ainsi possible d'effectuer des actions telles que

Lecture, Stop, Retour arrière, Avance rapide et Pause. L'application utilise cette interface pour contrôler le player. On ajoute cependant un bouton (Capture) qui prend l'image courante et la copie dans un bitmap résidant en mémoire.

Pourquoi utiliser un bitmap ? Vous pourriez vous contenter de copier l'image ailleurs dans la fiche principale ou dans une paintbox. Cependant, si on déplaçait une fenêtre au-dessus de l'image capturée, et si on l'en retirait, l'image serait effacée. En copiant l'image dans un bitmap, vous la stockez dans une partie de la mémoire que vous contrôlez totalement.

L'usage d'un bitmap complique cependant un peu notre programme. Où doit-on stocker ce bitmap ? Seules les procédures et les fonctions de la fiche ont besoin d'y accéder. Nous pouvons donc ajouter le bitmap à la partie `Private` de la déclaration de classe de `TForm`. Bien qu'il soit référencé dans la fiche, vous devez tout de même appeler la méthode `Create` et spécifier ses dimensions lorsque la fiche est chargée. Ce code peut être placé dans le Gestionnaire d'événements `OnCreate` de la fiche. De même, vous devez procéder à un nettoyage final en supprimant le bitmap lorsque la fiche se ferme. Remarquez qu'en stockant l'image dans un bitmap, vous ne l'affichez pas dans la fiche. Placez un composant `PaintBox` sur la fiche et utilisez son événement `OnPaint` pour copier le bitmap dans la `PaintBox` à chaque appel à l'événement `OnPaint`. Le Gestionnaire d'événements `OnPaint` est le suivant :

```
procedure TForm1.PaintBox1Paint(Sender: TObject);
begin
   PaintBox1.Canvas.CopyRect(Rect(0,0,200,200),ImgBitmap
Canvas,Rect(0,0,200,200));
end;
```

Ainsi, chaque fois que le système d'exploitation détermine que l'image doit être redessinée, il appelle la procédure `PaintBox1Paint` pour le faire.

Utiliser les méthodes pour contrôler le lecteur multimédia

Il arrive que vous souhaitiez utiliser des fonctions multimédias dans une application, sans pour autant afficher le composant lecteur multimédia. Il vous suffit d'affecter `False` à la propriété `Visible` du composant media player, et d'utiliser toutes ses fonctions par le biais des méthodes du composant. Il existe des méthodes correspondant à tous les boutons de l'interface du lecteur multimédia, ainsi que des méthodes offrant des fonctions qui ne sont pas prises en charge par l'interface seule. C'est le cas, par exemple, de la méthode `Open`, qui permet d'ouvrir un fichier.

Il est courant de cacher le composant `media player` lorsque vous souhaitez afficher un clip vidéo dans une application mais ne voulez pas dérouter l'utilisateur ou encombrer l'interface. Imaginez, par exemple, une application de livre de recettes. L'utilisateur pourrait cliquer sur

un bouton pour voir un clip vidéo dans lequel un grand cuisinier prépare un plat. Dans ce cas, il serait inutile de faire figurer l'interface toute entière, seul le clip vidéo importe à l'utilisateur.

Répondre aux événements du Lecteur multimédia

Bien que le composant Lecteur multimédia comporte de nombreuses propriétés et méthodes, il compte relativement peu d'événements. Les événements qui sont fournis permettent de s'immiscer dans l'interface utilisateur fournie et d'être prévenu lorsqu'une tâche multimédia s'achève. Il est ainsi possible d'exécuter un code spécifique lorsque l'utilisateur clique sur l'un des boutons, et d'indiquer au lecteur multimédia s'il doit exécuter la tâche demandée.

Stockage des fichiers vidéo

Nouveau

Dans les exemples précédents, vous avez vu comment utiliser des fichiers AVI dans vos applications pour utiliser de la vidéo. Mais qu'est-ce qu'un fichier vidéo précisément, et comment fonctionne-t-il ? Le cerveau humain voit un mouvement dans une succession rapide d'images légèrement modifiées. Dans une vidéo, chaque image est légèrement différente de la précédente. Pour que la vidéo, ou l'animation, soit fluide, la vitesse idéale se situe aux alentours de 30 images par seconde. Pour une vitesse supérieure, la différence n'est guère perceptible et pour une vitesse inférieure, l'image semble sautiller.

Faisons un petit calcul. Un bitmap plein écran fait plusieurs centaines de kilo-octets. Si vous deviez stocker chaque image d'une vidéo sous forme de bitmap, vous auriez besoin de capacités de stockage gigantesques. En utilisant cette méthode, vous ne pourriez en effet stocker que 72 secondes de vidéo sur un CD-ROM... S'il n'existait que cette méthode, toute application multimédia serait impraticable. La solution réside dans la compression.

Plutôt que de se lancer dans de longues explications mathématiques décrivant la compression vidéo d'un format particulier (AVI ou MPEG par exemple), nous allons nous contenter de présenter, dans les grandes lignes, l'une des techniques mises à l'œuvre. Dans certains dessins animés japonais on voit l'image s'arrêter alors que seule la bouche du personnage continue de bouger. Les animateurs ont vite compris qu'il est beaucoup plus facile de ne changer qu'une partie de la scène plutôt que la totalité de la scène. La compression vidéo s'appuie sur un principe similaire. Lorsqu'une image est capturée, le matériel ou le logiciel de compression prend une décision : "Est-ce que je peux stocker cette image en utilisant moins de place en n'enregistrant que les parties différentes de l'image précédente, ou est-ce que j'enregistre l'image dans son intégralité ?". La plupart du temps, il est plus facile de n'enregistrer que les parties qui ont changé. Cependant, dans certains cas, lorsque le plan change par exemple, la description des divers changements prendrait plus de place que le simple enregistrement de l'image elle-même. Les techniques de stockage de vidéo ont fait bien des progrès. Il est désormais possible de stocker un long métrage sur un CD-ROM standard.

Techniques d'animation en Delphi

Lorsque vous utilisez des animations ou des graphiques en mouvement, il est important que le mouvement semble fluide. Malheureusement, créer une image en dessinant une forme, puis l'effacer pour en dessiner une autre, produit une animation dont le scintillement irrite la vue.

Doubles tampons utilisant les services Windows standard

Nouveau

Un double tampon est un ensemble de surfaces de dessin. Une surface est affichée et l'autre est utilisée comme surface de dessin. Lorsque le dessin est achevé dans le tampon de dessin, les tampons permutent de telle sorte que celui qui était auparavant caché est maintenant affiché, ou encore le tampon caché est rapidement copié dans le tampon d'affichage.

Il existe plusieurs méthodes pour construire un double tampon en Delphi. Une d'elles consiste à créer un bitmap en mémoire et à l'utiliser comme tampon temporaire. Dessinez l'image sur le bitmap et lors du dessin de chaque image, utilisez une méthode `CopyRect` pour copier l'image bitmap dans le tampon d'affichage.

Un cube virevoltant

Les principes de l'animation en Delphi maintenant assimilés, vous pouvez construire un exemple interactif. Cet exemple montrera qu'il est possible d'obtenir de beaux effets d'animation en Delphi sans pour autant utiliser les techniques graphiques accélérées disponibles dans OpenGL ou DirectX. Cette application affiche un cube dans une fiche. L'utilisateur peut définir la vitesse de rotation du cube par rapport aux axes X, Y et Z. La Figure 10.13 vous montre l'application en action.

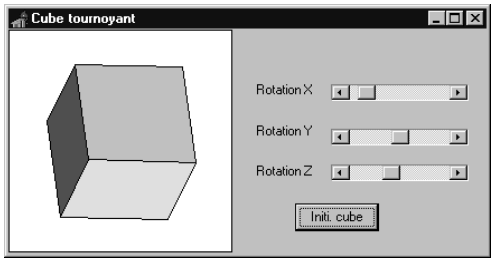


Figure 10.13
Le cube virevoltant.

Cet exemple illustre les principes évoqués dans cette section tout en introduisant quelques notions de graphisme en trois dimensions. La première étape de la création du cube consiste à savoir comment dessiner chaque image. Le cube est composé de six faces et de huit points. Vous définissez la position de chaque point dans l'espace à trois dimensions en stockant une coordonnée (X, Y, Z) dans un enregistrement appelé TDPoint (Point à Trois Dimensions).

Vous assurez le suivi de tous ces points en utilisant un tableau d'enregistrements. Ce tableau est défini de la manière suivante :

```
Pnts : array[1..8] of TDPoint;   {Image initiale }
```

Listing 10.10 : Un cube virevoltant

```
unit unitSpinCube;
{************************************************}
{* Le cube 3d en rotation - Un exemple complet     *}
{* d'animation en Delphi. Cette application        *}
{* affiche un cube en rotation. L'utilisateur  peut*}
{* ajuster la vitesse de rotation selon chaque     *}
{* axe.                                           *}
{************************************************}

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs,
  ExtCtrls, StdCtrls;

type
  TForm1 = class(TForm)
    ZRot: TScrollBar;
    YRot: TScrollBar;
    XRot: TScrollBar;
    ResetCube: TButton;
    Timer1: TTimer;
    Label1: TLabel;  {Labels utilisés pour faire des barres de défilement}
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure Timer1Timer(Sender: TObject);
    procedure ResetCubeClick(Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
```

```
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
Type
{permet de créer une matrice de rotation }
  Matrix  =  array[0..3,0..3] of Extended;
{Structure stockant un point 3D }
  TDPoint =  record
     X : Extended;
     Y : Extended;
     Z : Extended;
  end;

var
  DoubleBuffer : TBitmap; {on dessinera là-dedans avant d'effectuer un
CopyRect}
  BlankBuffer  : TBitmap; {Bitmap qui stocke le fond }
  PntsOut :array[1..8] of TDPoint; {points en rotation }
  TPPnts : array[1..8] of TPoint;  {représentation 2D des points}
  Pnts : array[1..8] of TDPoint;   {Image initiale }
  XAng,YAng,ZAng : Extended;

{*******************************************************}
{On crée un tableau (ou matrice) qui définit la rotation selon des angles
passés en radians. }
{*******************************************************}
procedure matrixRotate(var m:Matrix;
                       x,y,z : Extended);
var
    sinX, cosX,
    sinY, cosY,
    sinZ, cosZ:Extended; {on le stocke ici pour ne calculer qu'une fois }
    C1,C2 : integer; {pour les boucles }

begin
    sinX := sin(x); {un peu de géométrie...}
```

```
    cosX := cos(x);
    sinY := sin(y);
    cosY := cos(y);
    sinZ := sin(z);
    cosZ := cos(z);
    for C1 := 0 to 3 do  {matrice Identité }
     for C2 :=0 to 3 do
       if C1 = C2 then
         M[C1,C2] := 0
        else
         M[C1,C2] := 1;
    M[0,0] :=  (cosZ * cosY);
    M[0,1] :=  (cosZ * -sinY * -sinX + sinZ * cosX);
    M[0,2] :=  (cosZ * -sinY * cosX + sinZ * sinX);
    M[1,0] :=  (-sinZ * cosY);
    M[1,1] :=  (-sinZ * -sinY * -sinX + cosZ * cosX);
    M[1,2] :=  (-sinZ * -sinY * cosX + cosZ * sinX);
    M[2,0] :=  (sinY);
    M[2,1] :=  (cosY * -sinX);
    M[2,2] :=  (cosY * cosX);
end;

{On applique la matrice de rotation à un point 3D pour obtenir un autre
point 3D }
procedure ApplyMatToPoint(PointIn : TDPoint;
                          var pointOut:TDPoint;mat : Matrix);
var
    x, y, z : Extended;

begin
    x :=(PointIn.x * mat[0,0]) + (PointIn.y * mat[0,1]) +
        (PointIn.z * mat[0,2]) + mat[0,3];
    y := (PointIn.x * mat[1,0]) + (PointIn.y * mat[1,1]) +
        (PointIn.z * mat[1,2]) + mat[1,3];
    z := (PointIn.x * mat[2,0]) + (PointIn.y * mat[2,1]) +
        (PointIn.z * mat[2,2]) + mat[2,3];
    PointOut.x :=  x;
    PointOut.y :=  y;
    PointOut.z :=  z;
end;
```

```
{Notre cube a huit points, correspondant aux 8 sommets. Nous allons définir
les coordonnées de chaque point. Le centre du cube est en (0,0,0)  }
procedure InitCube;
begin
  Pnts[1].X := -50;
  Pnts[1].Y := -50;
  Pnts[1].Z := -50;
  Pnts[2].X := 50;
  Pnts[2].Y := -50;
  Pnts[2].Z := -50;
  Pnts[3].X := 50;
  Pnts[3].Y := 50;
  Pnts[3].Z := -50;
  Pnts[4].X := -50;
  Pnts[4].Y := 50;
  Pnts[4].Z := -50;
  Pnts[5].X := -50;
  Pnts[5].Y := -50;
  Pnts[5].Z := 50;
  Pnts[6].X := 50;
  Pnts[6].Y := -50;
  Pnts[6].Z := 50;
  Pnts[7].X := 50;
  Pnts[7].Y := 50;
  Pnts[7].Z := 50;
  Pnts[8].X := -50;
  Pnts[8].Y := 50;
  Pnts[8].Z := 50;
end;

{La fonction qui suit renvoie true si la somme des paramètres est supérieure
à zéro et false sinon. Cette fonction permet de déterminer quelles sont les
faces du cube à cacher}
function ShowSide(V1,V2,V3,V4 : Extended) : Boolean;
begin
  if (V1+V2+V3+V4) > 0 then
    ShowSide := TRUE
   else
    ShowSide := FALSE;
end;
```

```
{On utilise un tampon double. Cette fonction détermine si une face est
visible. Si c'est le cas, elle dessine la représentation 2D de notre tampon
bitmap avec pour couleur de remplissage celle passée }
procedure AddSide(P1,P2,P3,P4:Integer;SideColor : TColor);
begin
 if ShowSide(PntsOut[P1].Z,PntsOut[P2].Z,
             PntsOut[P3].Z,PntsOut[P4].Z) then
  begin
    DoubleBuffer.Canvas.Brush.Color := SideColor;
    DoubleBuffer.Canvas.Polygon([TPPnts[P1],TPPnts[P2],
                                 TPPnts[P3],TPPnts[P4],TPPnts[P1]]);
  end;
end;

procedure TForm1.FormCreate(Sender: TObject);
{Lorsque la fiche se charge, on crée et on initialise notre bitmap de fond
et on initialise notre bitmap double tampon}
begin
   DoubleBuffer := TBitmap.Create;
   DoubleBuffer.Height := 200;
   DoubleBuffer.Width := 200;
   BlankBuffer := TBitmap.Create;
   BlankBuffer.Height := 200;
   BlankBuffer.Width  := 200;
   BlankBuffer.Canvas.Brush.Color := clWhite;
   BlankBuffer.Canvas.rectangle(0,0,200,200);
   InitCube();
   XAng := 0;
   YAng := 0;
   ZAng := 0;
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
{Quand tout est fini on fait le ménage en libérant les bitmap }
begin
   BlankBuffer.Free;
   DoubleBuffer.Free;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
{La procédure principale qui dessine le cube. Cette procédure est appelée
```

```
par un timer qui exécute sa fonction OnTimer toutes les 20 ms}

var
  M : Matrix; {La matrice de rotation }
  Count2 : Integer; {pour boucler à travers les points }

begin
    XAng := XAng + XRot.Position;  {on ajuste l'angle de rotation }
    YAng := YAng + YRot.Position;  {en agissant sur les barres de }
    ZAng := ZAng + ZRot.Position;  {défilement }
    {On ajuste en degré et on construit la matrice de rotation }
    matrixRotate(M,(PI*XAng)/180,(PI*YAng)/180,(PI*ZAng)/180);
    {On parcourt tous les points et on effectue la rotation pour}
    {obtenir la représentation 2D }
    for Count2:= 1 to 8 do
    begin
      ApplyMatToPoint(Pnts[Count2],PntsOut[Count2],M);
      TPPnts[Count2] := Point(trunc(PntsOut[Count2].X+100),
                              trunc(PntsOut[Count2].Y+100));
    end;
{On efface le tampon double en effectuant un CopyRect sur le fond }
    DoubleBuffer.Canvas.CopyRect(RECT(0,0,200,200),
                                 BlankBuffer.Canvas,RECT(0,0,200,200));
{On construit le cube en appelant AddSide pour chacune des six faces }
    AddSide(1,2,3,4,clBlue);
    AddSide(5,6,7,8,clRed);
    AddSide(1,2,6,5,clYellow);
    AddSide(2,3,7,6,clGreen);
    AddSide(3,4,8,7,clPurple);
    AddSide(4,1,5,8,clSilver);
{On copie le tampon double sur la fiche }
    Form1.Canvas.CopyRect(RECT(0,0,200,200),
                          DoubleBuffer.Canvas,RECT(0,0,200,200));
end;

procedure TForm1.ResetCubeClick(Sender: TObject);
begin
  XAng := 0;
  YAng := 0;
  ZAng := 0;
```

```
end;
end.
```

Analyse

Les points sont initialisés dans la procédure InitCube (voir Listing 11.6).Ces valeurs ne changeant jamais, cela n'est nécessaire qu'une seule fois. A chaque image, on détermine la position du cube et on calcule un nouvel ensemble de points prenant en compte les coordonnées du cube en rotation. Cela est mené à bien en créant une matrice de rotation, puis en appelant `ApplyMatToPoint` *pour chacun des huit points. Vous obtenez alors un nouvel ensemble de points dans l'espace. Il vous suffit alors de vous débarrasser de la valeur en Z et pour obtenir vos coordonnées X et Y pour les huit sommets.*

Vous pouvez avoir recours à une petite astuce pour déterminer les faces cachées du cube. Si un cube est situé en (0,0,0), une face apparaît si la somme de ses valeurs en Z est supérieure à 0. Cette astuce n'est valable que pour certains types d'objets, et généralement la gestion des faces cachées est bien plus complexe. Dans notre cas, elle est assurée dans la fonction simple `ShowSide`.

Maintenant que vous savez quoi dessiner et avec quelles coordonnées, il ne vous reste plus qu'à tout animer de manière fluide. Chaque image est déclenchée par un événement OnTimer. La procédure lit les composants curseurs pour déterminer l'incrémentation de chaque angle de rotation. Pour dessiner l'image, utilisez un bitmap en mémoire et initialisez-le en copiant un autre bitmap. On utilise le deuxième bitmap plutôt qu'une méthode Rectangle car CopyRect est plus rapide. Vous déterminez ensuite quelles sont les faces devant s'afficher et vous les dessinez sur le bitmap en mémoire en utilisant une méthode polygone. Chaque face est dessinée à l'aide d'une couleur distincte pour rendre l'effet d'animation encore plus spectaculaire. Enfin, vous utilisez un CopyRect pour copier le bitmap en mémoire sur la fiche principale. C'est un exemple intéressant qui peut être facilement modifié et étendu.

Pour plus de performances : DirectX et OpenGL

L'exemple précédent a montré comment créer des graphismes en trois dimensions à l'aide de composants Delphi standard. Un des problèmes posés par cette technique est qu'il n'est pas possible de tirer parti du matériel conçu spécifiquement pour le rendu 3D (alors que la plupart des nouvelles cartes graphiques en sont capables). De plus, cette technique vous oblige à effectuer vous-même toutes les tâches automatisées par un moteur graphique 3D, comme la suppression des surfaces cachées et les ombrages.

OpenGL et Direct3D (une des composantes de DirectX) sont spécifiquement conçus pour effectuer un rendu graphique optimisé tirant parti du matériel 3D. Direct3D se fonde sur le modèle objet de composant (COM) et offre les meilleures performances. OpenGL définit une interface spécialisée de commande graphique. OpenGL et Direct3D sortent cependant du cadre de cet ouvrage ; nous vous invitons à consulter des ouvrages consacrés à ce sujet.

Récapitulatif

Le développement graphique est l'un des domaines les plus courus du moment. Delphi est un outil parfaitement adapté pour exploiter les fonctionnalités graphiques de Windows 95 et Windows NT. La surface primaire de dessin en Delphi est le canevas. Ce canevas permet à une application de dessiner sur une fiche, une paintbox ou un bitmap. Lors de cette journée, nous avons vu différentes façons de manipuler un canevas, plusieurs techniques de travail sur les images et les bitmap, ainsi que certains points théoriques sur le graphisme en général.

Le mot magique de *multimédia* est maintenant sur toutes les lèvres. En fait, le multimédia n'est autre que l'utilisation du son et de la vidéo dans des applications. Cette partie vous a montré comment utiliser l'appel API `PlaySound` pour ajouter des effets sonores à une application. Nous avons également vu le composant visuel Lecteur multimédia (media player) pouvant lire toutes sortes de fichiers multimédias, fichiers vidéo inclus. Le composant Lecteur multimédia est très configurable et ses usages sont multiples.

Enfin, nous avons parlé de l'animation. Le Lecteur multimédia est l'outil de choix pour la lecture de vidéos préenregistrées, mais il n'est pas adapté à la création d'animations évolutives. Vous avez pu voir comment améliorer les performances en tirant parti d'algorithmes plus ou moins efficaces. Cette journée d'apprentissage s'est terminée sur une animation de cube en rotation qui a fait la démonstration de la puissance des méthodes de Delphi.

Atelier

L'atelier vous donne trois façons de vérifier que vous avez correctement assimilé le contenu de cette journée. La section Questions - Réponses vous propose des questions couramment posées, ainsi que leurs réponses. Les réponses aux questions de la section Questionnaire se trouvent dans l'Annexe A, et les Exercices vous permettent de mettre en pratique ce que vous venez d'apprendre. Tentez dans la mesure du possible de vous appliquer à chacune des trois sections avant de passer à la suite.

Questions - Réponses

Q Comment mon application peut-elle utiliser des API graphiques Win32 en conjonction avec des composants graphiques en Delphi ?

R La plupart des composants et objets Delphi sont dotés de propriétés que vous pouvez utiliser pour manipuler les handles sous-jacents. Ainsi, la classe `TBitmap` contient une propriété `Handle` à laquelle vous pouvez accéder pour effectuer des API de bitmap. De plus, le contexte de périphérique (DC) du canevas est disponible par le biais de cette même propriété `Handle`.

Q En quoi est-il plus intéressant de dessiner un bitmap hors de l'écran avant de le copier sur l'affichage ?

R Lorsque vous dessinez sur un bitmap hors de l'écran, l'image ne subit pas de scintillement. Vous pouvez conserver ainsi plusieurs images et les substituer rapidement les unes aux autres dans une partie de l'écran afin de créer une animation.

Q Est-il possible d'utiliser la MCI (interface de contrôle multimédia) sans utiliser le composant lecteur multimédia ?

R Oui, tous les appels API multimédias sont disponibles. Si vous disposez du code source du lecteur multimédia, vous verrez qu'il utilise lui-même l'API MCI.

Q `CopyRect` **a été utilisé pour déplacer rapidement une portion rectangulaire d'un canevas. Y a-t-il d'autres moyens pour déplacer rapidement des portions d'un canevas ?**

R Oui, vous pouvez accéder à la propriété `Handle` d'un canevas, qui est un handle vers son contexte de périphérique. Vous pouvez alors utiliser tous les appels GDI Win32 que vous souhaitez pour manipuler l'image. Il existe un appel qui permet de déplacer des données vers un parallélogramme, vous permettant également d'effectuer diverses opérations de masque.

Q Plusieurs threads peuvent-ils écrire sur le même canevas ?

R Oui. Vous devez appeler la méthode `Lock` sur le canevas avant de le manipuler, et la méthode `UnLock` lorsque vos opérations sont terminées.

Questionnaire

1. Dans l'exemple d'animation simple, que se passe-t-il si `pmCopy` est utilisé pour le mode de crayon au lieu de `pmNotXor` ?
2. Le clipping est utile pour circonscrire un dessin dans une portion donnée d'une fiche. Quels composants Delphi pourriez-vous utiliser pour découper ainsi un dessin selon une région de l'écran ?
3. Comment une application peut-elle utiliser une méthode pour charger un bitmap dans une image ? Comment copier un morceau d'un bitmap dans une paintbox ?
4. Comment PlaySound sait-il s'il doit attendre que le son ait été lu pour exécuter l'instruction suivante, ou s'il doit au contraire exécuter l'instruction suivante alors que le son continue à être lu ?
5. Quel est l'effet de l'appel :

```
MediaPlayer1.DisplayRect := RECT(50,50,200,200);
```

   sur un composant Lecteur multimédia ? Indice : il ne demande pas d'afficher les coordonnées dans la portion délimitée par (50,50) et (200,200).
6. Lorsque vous effectuez une animation, pourquoi n'est-il pas intéressant de dessiner une image, puis de l'effacer avant de dessiner la suivante ?

Exercices

1. Vérifiez que lorsqu'une fenêtre contenant un objet graphique ne se réaffiche pas d'elle-même après avoir été recouverte, l'image qui figure dans cette portion de la fenêtre est effacée.
2. Modifiez le programme MIXUP de façon que les pièces du bitmap nouvellement créé soient disposées dans un ordre spécifique plutôt qu'aléatoire (en inversant les X et les Y par exemple).
3. Ecrivez un programme qui joue constamment de la musique de fond.
4. Utilisez le composant lecteur multimédia pour lire un fichier vidéo. Laissez à l'utilisateur le choix de déterminer l'endroit où la vidéo sera jouée (dans sa propre fenêtre ou dans la fiche principale). Ajoutez des barres de défilement permettant à l'utilisateur de définir la taille de l'image.
5. Modifiez le programme de rotation de cube afin que le cube soit en rotation sur un fond en bitmap. Une astuce : remplacez le bitmap vierge utilisé pour nettoyer l'image par un vrai bitmap.

11

Les bases de données sous Delphi

Tout programmeur est tôt ou tard confronté au problème suivant :

L'application doit accéder à de gros volumes de données et les manipuler. Existe-t-il un moyen simple d'y parvenir ?

La réponse est oui. Les systèmes de bases de données sont des outils génériques de manipulation de données. Un moteur de bases de données fournit les mécanismes nécessaires pour manipuler et visualiser les données de la base. Sans les bases de données, les programmeurs seraient obligés d'écrire des routines complexes pour gérer les fichiers de façon efficace. En résumé, le programmeur devrait gérer et la base de données, et son moteur.

L'implémentation des bases de données dans Delphi est simple et efficace. Elle comporte des composants visuels permettant d'accéder à des tables et fournissent des méthodes pour manipuler les enregistrements. Cette partie explique comment créer vos propres bases de données en utilisant les outils fournis avec Delphi, et comment intégrer ces bases de données dans vos applications. Delphi peut aussi bien faire des manipulations complexes de bases de données que se comporter comme un simple terminal.

Nouveau

Un terminal de bases de données est un programme qui possède une interface simple d'emploi afin d'accéder aux données d'une base de données et les manipuler. Delphi comprend un outil rendant le développement d'applications quasi automatique en gérant l'essentiel des opérations à votre place. Cet outil s'appelle l'Expert fiche base de données. Il crée une application complètement fonctionnelle sans vous demander d'écrire une seule ligne de code.

Modèle de données relationnel

La plupart des bases de données récentes sont relationnelles. Une base de données relationnelle stocke l'information dans des tables logiques composées de lignes et de colonnes. Ces tables sont appelées des tables de bases de données. Examinons une table simple.

L'université de RAD (RADU) désire garder des enregistrements relatifs à ses étudiants. Pour cela, RADU organise ses données en table. Cette table figure au Tableau 11.1.

Tableau 11.1 : Données sous forme relationnelle

NoSS	Nom	Classe	Téléphone	Moyenne
2650392125351	Ada Smith	Deug1	01-43-23-45-64	13,4
1660221530400	Henry Ford	Deug1	01-51-25-45-21	12,8
1651102003020	Ragle Gumm	Deug1	04-15-94-52-46	13,8
1631203100235	Niklaus Wirth	Deug2	02-15-45-19-54	14,0

Nouveau

Les colonnes d'une table s'appellent des champs, et les lignes des enregistrements. Pour chaque enregistrement dans la table, il doit y avoir une valeur par champ.

Premiers pas : l'Expert fiche base de données

Delphi est fourni avec un puissant outil qui transcrit les champs d'une table en champs d'édition d'une fiche. Voici un exemple d'utilisation de cet outil pour créer une application en utilisant l'Expert de fiche. Cet exemple utilise l'une des bases de données fournies avec Delphi.

L'Expert fiche base de données

Il est aussi simple qu'amusant d'utiliser l'Expert fiche base de données de Delphi. Faites des choix et l'Expert génère automatiquement le code et les écrans à votre place. Suivez ces étapes pour créer votre premier écran de base de données :

1. Lancez Delphi.
2. Choisissez Base de données/Expert fiche... depuis le menu principal. La boîte de dialogue Expert fiche base de données apparaît alors (voir Figure 11.1).

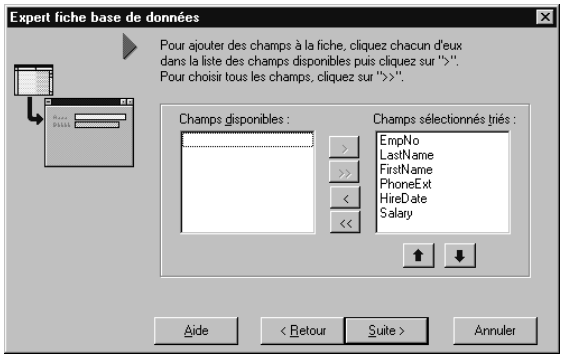
Figure 11.1
Sélection des options dans l'Expert fiche base de données.

3. Dans la première boîte de dialogue, sélectionnez Créer une fiche simple, et Créer une fiche utilisant des objets TTable. Cliquez sur Suivant.
4. Dans la boîte de dialogue suivante, sélectionnez DBDEMOS dans la section Lecteur (ou Alias). Cela fait apparaître une liste de tables dans la boîte de liste de gauche (voir Figure 11.2).

Figure 11.2
Choix des tables à utiliser.

5. Choisissez EMPLOYEE.DB. Cliquez sur Suivant.
6. Cliquez sur le bouton >> pour indiquer que vous voulez construire une fiche comportant tous les champs (voir Figure 11.3).

Figure 11.3
Choix des champs à utiliser.



7. Cliquez sur le bouton Suivant en acceptant toutes les valeurs par défaut jusqu'à la création de la fiche.
8. Choisissez Exécuter/Exécuter dans la barre de menu.

Félicitations ! Vous venez de créer votre première application de bases de données sous Delphi (voir Figure 11.4).

Comment ça marche ?

Cette application donne à l'utilisateur le plein pouvoir sur une table contenant des informations sur les employés. Pour manipuler les enregistrements, une barre de navigation est insérée en haut de l'application. Chaque bouton de cette barre exécute une manipulation de la base de données. La fonction de chacun des boutons est détaillée dans le Tableau 11.2.

Figure 11.4
L'application générée par l'Expert fiche base de données.

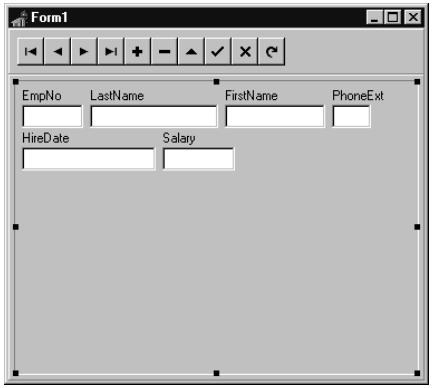


Tableau 11.2 : Les boutons de navigation de base de données

Icône	*Fonction*	**Description**
	Premier enregistrement	Se place sur le premier enregistrement de la table
	Précédent	Revient à l'enregistrement précédent.
	Suivant	Passe à l'enregistrement suivant.
	Dernier	Passe au dernier enregistrement de la table.
	Insérer	Insère un nouvel enregistrement à la position courante.
	Supprimer	Supprime l'enregistrement affiché.
	Editer	Permet de modifier les champs de l'enregistrement affiché.
	Valider	Valide les modifications apportées à l'enregistrement.
	Annuler	Annule les modifications apportées à l'enregistrement.

Icône	*Fonction*	**Description**
	Rafraîchir	Recharge tous les enregistrements courants et les autres. Cette fonction est surtout utile si plusieurs applications accèdent à la même base de données, les données de l'enregistrement courant pouvant être modifiées.

Choix de la base de données

L'une des fonctionnalités les plus puissantes du moteur de bases de données de Delphi est qu'il dispose d'une couche d'abstraction venant se placer entre les données sous-jacentes et les mécanismes de traitement. A la question : "Quelle base de données dois-je utiliser ?", il n'y a donc pas de réponse privilégiée, et le choix d'une base de données n'est pas définitif : vous pourrez changer le type de base de données sans modifier votre code.

Modèles de bases de données

Beaucoup d'applications utilisent des bases de données. Un centre des impôts et un garage n'utilisent pas les mêmes : le volume de données, les traitements qui leur sont appliqués, et le nombre d'applications différentes susceptibles de les exploiter diffèrent largement. D'autre part, certains problèmes requièrent des bases de données utilisées par plus d'un programme et plus d'une personne en même temps. Que se passerait-il, par exemple, si deux programmes essayaient de modifier en même temps le même enregistrement ? Quelles sont les solutions pour qu'une application ne puisse accéder qu'à une partie d'une base de données ? Ces deux problèmes sont appelés problèmes d'accès concurrent et problèmes de sécurité. En fonction d'eux, il convient de choisir différents types de bases de données.

La puissance, la complexité et les fonctionnalités d'un système de bases de données sont souvent liées à l'infrastructure physique sous-jacente. Nous allons maintenant explorer les trois principales catégories de bases de données : locale, à fichiers partagés, et Client/Serveur. Nous verrons également la structure de base de données multiliaison fournie avec Delphi CS.

Les sections suivantes décrivent les différents types de modèles de base de données :

- Locale
- A fichiers partagés
- Cllient/Serveur
- Multiliaison

Bases de données locales

Les bases de données locales sont les plus simples à manipuler, car un grand nombre de problèmes ne se posent plus. Les données sont stockées dans la machine locale, et le moteur est également sur cette machine. Il est donc impossible que deux utilisateurs essaient simultanément d'accéder à la base de données : les problèmes de concurrence disparaissent. Généralement, une telle base de données demande rarement des calculs complexes qui s'effectueraient au détriment du confort de l'utilisateur. Les bases de données locales sont utiles pour les applications distribuées à un grand nombre d'utilisateurs, chacun maintenant ses propres données de façon indépendante. Par exemple, une application destinée à suivre le nombre de kilomètres parcourus par une voiture afin de rembourser les frais de carburant pourrait être développée selon ce modèle. Chaque personne utilisant l'application gérerait sa propre consommation sur son ordinateur : personne n'ayant besoin de connaître les consommations des autres, une base de données locale est bien adaptée. Il existe un grand nombre de systèmes de gestion de bases de données locales. Les bases de données Paradox et dBASE peuvent être créées et manipulées en standard avec Delphi. Ces systèmes de bases de données peuvent s'utiliser en local ou en fichiers partagés.

Bases de données partagées

Une base de données à fichiers partagés se comporte presque comme une base de données locale, excepté le fait que plusieurs programmes peuvent y accéder au travers d'un réseau. La base de données peut donc être manipulée par plusieurs personnes en même temps. Considérons par exemple la base des employés d'une entreprise. Si un administrateur change le salaire d'un employé, l'application de paie sera immédiatement avertie de la modification. Cependant, si les données de la base sont distantes, le moteur de bases de données reste local. Un autre avantage des bases de données à fichiers partagés est qu'elles ne demandent pas de connaissance *a priori* du type de réseau utilisé. La base de données se moque de savoir si le système d'exploitation est Banyan, Novell, ou Microsoft NT, par exemple, parce qu'elle considère les données comme un simple fichier. Cette formule trouve cependant ses limites dès que le nombre d'accès concurrents ou la charge de calcul augmente. Dans ces situations, on se tourne vers les bases de données Client/Serveur.

Bases de données Client/Serveur

La solution la plus sophistiquée pour l'accès concurrent à une base de données est le modèle Client/Serveur. Dans ce cadre, une machine dédiée, appelée serveur, est conçue pour gérer les accès d'un groupe de clients à la base. Prenons par exemple la base de données des impôts sur le revenu, et supposons qu'une application demande l'ensemble des numéros de sécurité sociale pour lesquels l'impôt a été compris entre 10 000 et 800 000 francs cette année. Une telle requête sur un système à fichiers partagés bloquerait la base de données et tout le système pour un certain temps. Dans le cadre Client/Serveur, le client effectue la requête auprès du serveur. C'est alors au Client de décider s'il préfère attendre la réponse ou bien faire autre chose pendant l'exécution de la requête. De son côté, le Serveur a été optimisé pour traiter les requêtes le plus rapidement possible. Cependant, l'architecture Client/Serveur a aussi un inconvénient :

ces solutions reviennent plus cher que celles à fichiers partagés, et demandent une connaissance préalable de l'implémentation réseau utilisée (par exemple TCP/IP). Bien qu'il soit possible d'installer plusieurs couches réseau sur une même machine, on perd de la souplesse.

Un nouveau paradigme : les bases de données multiliaisons

Voici une nouvelle façon d'aborder la manipulation des bases de données dans un réseau. Cette approche est qualifiée de système multiliaisons. Elle se compose d'une combinaison de clients "légers", de serveurs d'application, et de serveurs de bases de données. Le résultat est un système permettant la gestion des pertes et des réplications de données. Nous ne donnerons pas plus de détails sur ce nouveau modèle puisqu'il n'est fourni que dans l'édition CS de Delphi. Si vous avez la version CS, référez-vous à l'aide en ligne.

Lors d'un développement sous Delphi ne demandant pas une compatibilité avec un système de bases de données préexistant, le développeur choisira très probablement dBASE, Paradox, ou Access, ces trois systèmes étant très bien intégrés dans Delphi. Le Tableau 11.3. donne quelques-unes des caractéristiques de ces trois systèmes :

Tableau 11.3 : Comparaison des tables Paradox, dBASE et Access

Attribut	*Paradox*	*dBASE*	*Access*
Nombre d'enregistrements	2 milliards	1 milliard	2 milliards
Champs par table	255	1024	
Caractères par champ	ns	256	ns
Octets par enregistrement	32750	32767	32767

Comme vous le voyez, ces trois systèmes sont extrêmement puissants et capables de gérer plus de données que vos applications n'en demandent. Paradox et Access offrent un peu plus de souplesse au niveau de la base de données. Cela signifie que les tables savent stocker autre chose que les données brutes. Paradox comprend notamment des critères de validation au niveau de la table, ainsi que des mots de passe. Ces caractéristiques prennent tout leur intérêt si l'application Delphi n'est pas la seule à accéder aux données.

Choisir le bon modèle de bases de données pour une application est souvent une tâche difficile. Fort heureusement, le moteur de bases de données Borland est suffisamment souple pour changer de modèle de base de données sans grands efforts. Si cela n'offre pas assez de fonctionnalités, le moteur de bases de données peut communiquer avec ODBC, lui-même pouvant communiquer avec la grande majorité des systèmes de gestion de bases de données (SGBD) du marché. Nous parlerons plus loin d'ODBC, mais pour le moment, voyons comment créer une nouvelle table en utilisant le Module base de données.

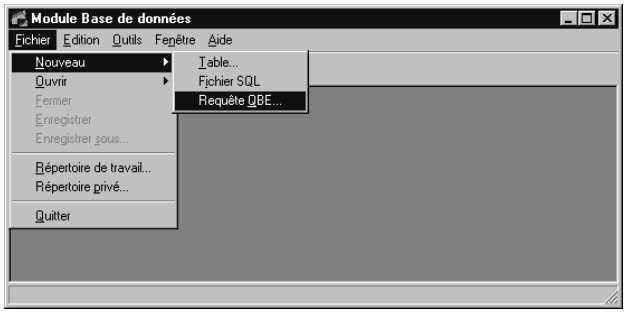
Alias

Les bases de données peuvent être gérées de différentes façons. Sous Microsoft Access, plusieurs tables sont stockées dans un seul fichier. Paradox et dBASE utilisent un fichier par table. Un système Client/Serveur comme Sybase ou Microsoft SQL server stocke toutes les données sur une machine séparée et communique avec les clients à travers un langage dédié appelé SQL (nous en parlerons plus loin). En plus des méthodes de stockage des données, certaines bases de données ont besoin d'informations supplémentaires. La tâche du moteur de bases de données est donc complexe, les différents systèmes de bases de données demandant des informations légèrement différentes. Une base de données Paradox, par exemple, voudra savoir quel répertoire représente la base de données, alors qu'une base de données Sybase pourra demander une adresse réseau pour le serveur, un nom de base de données, un nom d'utilisateur, et un mot de passe. La solution à ce problème repose sur les *alias*. Un alias contient toutes les informations nécessaires pour accéder à la base de données. Lors de la création d'un alias Paradox, il suffit de fournir le répertoire contenant la base de données.

Créer une nouvelle table avec le module base de données

Maintenant que vous maîtrisez les différents éléments de base d'une table, vous allez en créer un exemple simple. Le Module base de données Borland (voir Figure 11.5) peut être lancé depuis le menu Outils de Delphi. Choisissez cette option pour lancer le Module base de données.

Figure 11.5
Le Module base de données Borland.

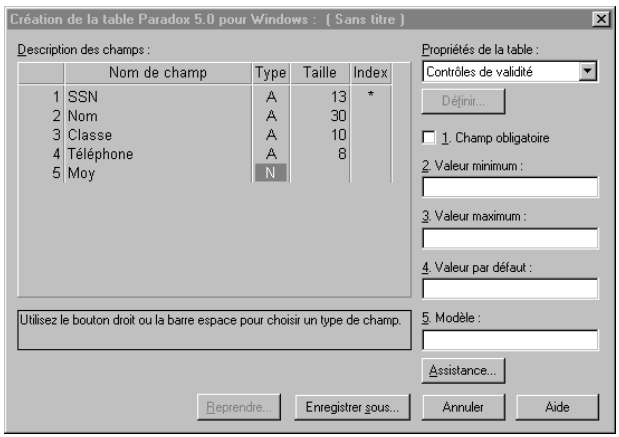


Création d'une nouvelle table

La création de tables sous Delphi est simplifiée par l'utilisation du Module base de données. Suivez ces instructions pour créer les tables et les champs :

1. Choisissez Fichier/Nouveau/Table dans la barre de menu. Une boîte de dialogue apparaît, vous demandant quel type de table vous désirez créer. Pour de nouvelles applications, Paradox est très probablement le meilleur moyen de travailler. Choisissez Paradox 7, et cliquez sur OK. La boîte de dialogue de la Figure 11.6 apparaît.

Figure 11.6
Définition des champs dans le Module base de données.



2. La première chose à faire est de décider quelles données vous voulez stocker dans votre table et quel est leur type. Pour cet exemple, considérons la table des enregistrements relatifs aux étudiants de RADU. Chaque champ doit être ajouté dans la zone description des champs de la boîte de dialogue. Pour ajouter le premier champ, tapez **SSN** dans le champ `Nom de champ`. Vous indiquez ainsi que vous voulez créer un nouveau champ appelé SSN.
3. L'information suivante est le type de données que vous allez stocker dans votre table. Les numéros de sécurité sociale sont stockés sous la forme d'une chaîne de caractères.Faites un clic droit dans le champ Type pour afficher les types disponibles. Pour une chaîne, Paradox utilise le type Alphanumérique. Choisissez Alphanumérique dans la liste de types affichée.
4. Pour certains types, une taille doit être spécifiée. Cela indique à la base de données la plus grande quantité de données pouvant être stockée dans le champ. Pour un numéro de sécurité sociale, indiquez 13. Certains types de données comme les nombres n'utilisent pas le champ de taille, et le Module base de données le rend inaccessible dans ce cas.
5. La dernière information indique si le champ sera ou non un index. S'il est un index, ce champ est unique pour chaque enregistrement, c'est-à-dire que deux enregistrements différents ne pourront pas avoir la même valeur pour ce champ. L'utilisation des index dans les bases de données est capitale, mais nous y reviendrons. Pour l'instant, indiquez que le champ SSN est un index, puisque chaque étudiant a un numéro de sécurité sociale différent des autres.

6. Pour définir le champ suivant de la table, appuyez sur Entrée. Continuez à définir les champs en utilisant les informations fournies dans le Tableau 11.4.

Tableau 11.4 : Définition et attributs d'une table simple

Nom de champ	*Type*	*Taille*	*Index*
SSN	A	13	*
Nom	A	30	
Classe	A	10	
Téléphone	A	10	
Moy	N		

7. Cliquez sur le bouton Enregistrer sous... pour sauvegarder les informations.
8. Dans la boîte de dialogue Enregistrement de la table, changez le lecteur (ou Alias) pour indiquer DBDEMOS. Tapez **STUDINFO.DB** dans la case Nouveau nom de fichier. Cliquez sur OK pour sauvegarder la table. C'est terminé, vous avez créé une nouvelle table.

Pourquoi utiliser des index ?

Pourquoi avoir défini SSN comme étant un index ? Les index aident le moteur de base de données à travailler efficacement. Il est utile de pouvoir distinguer chaque enregistrement des autres à l'intérieur d'une table. Paradox impose que tous les champs d'index soient définis en premier dans la définition des champs. Bien que la plupart des systèmes de bases de données vous dispensent d'avoir des index, c'est une bonne habitude à prendre que d'avoir au moins un index par table. Cependant, comment faire si la table sur laquelle vous travaillez n'a pas de candidat naturel au rôle d'index ? Cela signifie qu'il n'y a pas de champ qui soit unique en fonction de l'enregistrement. Regardons par exemple le Tableau 11.5, qui pourrait être utilisé pour une base de données de recettes de cuisine.

Tableau 11.5 : Ingrédients d'une base de données de recettes

Recette	**Ingrédient**
`Macaroni`	Macaroni
`Macaroni`	Cheddar
`Macaroni`	Lait
`Thon`	Thon
`Thon`	Cheddar

Vous ne pouvez pas utiliser Recette comme index car il y a plusieurs ingrédients par recette. Le terme Ingrédient ne convient pas mieux, puisqu'un ingrédient peut apparaître dans plusieurs recettes. Fort heureusement, la plupart des bases de données savent comment contourner le problème grâce à des types spécialisés qui sont gérés automatiquement par le moteur de base de donnés, garantissant que ce champ sera unique. Sous Paradoxe, ce type de données est appelé Incrémentation Auto. En définissant un champ de type Incrémentation Auto, vous créez un champ dont la valeur augmente de 1 à chaque nouvelle entrée, ce qui lui permet de servir d'index. Le Tableau 11.6 montre comment ajouter un champ d'index à vos recettes de cuisine.

Tableau 11.6 : Création d'un index dans la base de données recettes

Nom du champ	*Type*	*Taille*	*Index*
RecetteIndex	+		*
Recette	A	30	
Ingrédient	A	30	

Quand des données sont ajoutées à la table, chaque enregistrement se voit attribuer une valeur unique. Notre table ressemble maintenant à celle du Tableau 11.7.

Tableau 11.7 : Ingrédients d'une base de données de recettes

RecetteIndex	*Recette*	**Ingrédients**
1	Macaroni	Macaroni
2	Macaroni	Cheddar
3	Macaroni	Lait
4	Thon	Thon
5	Thon	Cheddar

Quelques informations devraient êtres ajoutées afin de pouvoir réellement travailler sur cette base de données, mais le modèle que nous avons créé est une bonne base de départ. Nous verrons plus loin comment augmenter les performances de la base de données et s'assurer que seules des informations valides peuvent y figurer.

Accéder à une table depuis Delphi

Nous vous avons montré la méthode la plus simple pour créer une application Delphi qui accède à une table : l'Expert Fiche de Delphi. Examinons maintenant de plus près ce qu'il se passe

derrière la scène et quelles étapes sont nécessaires pour construire une application en partant de zéro.

Le moyen le plus simple d'accéder à et de manipuler des bases de données sous Delphi consiste à utiliser les composants visuels fournis. Les composants visuels de bases de données sont stockés dans deux onglets de la palette de composants : les onglets AccèsBD et ContrôleBD. Les composants de l'onglet AccèsBD sont utilisés pour indiquer à Delphi quelles tables et quelles fonctions doivent être utilisées, alors que l'onglet ContrôleBD recense les composants visuels qui peuvent afficher les données d'une base de données ou fournir une interface pour manipuler les données (ajouter, insérer, modifier).

Indiquer une table à Delphi : le composant TTable

Pour utiliser une table, vous devez tout d'abord indiquer à Delphi que vous désirez travailler avec. Placez sur la fiche un objet Table accédant aux données de la table. La feuille de propriétés comprend deux propriétés nommées `DatabaseName`, et `TableName`. Ces deux propriétés sont indispensables pour indiquer à Delphi comment accéder à la table. La propriété `DatabaseName` correspond à l'alias. La propriété `TableName` indique quelle table de la base doit être utilisée.

Accéder à la table d'informations sur les étudiants

La première étape lors de la création d'une application destinée à accéder à la base de données que vous avez créée précédemment dans cette section consiste à placer un composant `TTable` sur votre fiche et à configurer les propriétés `DatabaseName` et `TableName`.

1. Créez une nouvelle application.
2. Placez un composant `TTable` sur la fiche. Le composant `TTable` se trouve dans l'onglet AccèsBD de la palette de composants.
3. Dans la feuille de propriétés, mettez `DatabaseName` à DBDEMOS. DBDEMOS est l'alias dans lequel nous avons sauvegardé la table STUDINFO.DB.
4. Après avoir défini la propriété `DatabaseName`, la propriété `TableName` indique les tables disponibles pour cet alias. Choisissez STUDINFO.DB comme valeur pour TableName.
5. Mettez la propriété `Active` à True. Cela a pour effet d'ouvrir la table lorsque l'application est lancée. Si vous laissez cet indicateur à False, l'application ne pourra pas accéder à la table tant que vous ne l'aurez pas mise à `True` durant l'exécution.
6. Donnez la valeur `StudInfo` au champ `Name`.

Deux autres propriétés intéressantes lors de l'exécution sont `ReadOnly` et `Exclusive`. La première vous permet de spécifier que la table n'est accessible qu'en lecture, et qu'il est donc impossible de la modifier. La seconde vous assure un accès exclusif à la table en empêchant d'autres applications d'y accéder. Nous avons donc fourni les informations nécessaires pour accéder à la table, mais nous ne pourrons manipuler les données qu'après avoir associé un DataSource à l'objet table.

Interfaçage des données : le composant TDataSource

Delphi peut accéder aux informations de bases de données à travers une série de composants appelés les composants de bases de données, comme par exemple le composant `TTable`. Les contrôles orientés données sont des contrôles qui affichent et manipulent les données d'une base à laquelle on accède depuis Delphi. Pour fournir une couche d'abstraction aux contrôles, il existe le composant `TDataSource`. Le `TDataSource` permet également de savoir si des changements ont eu lieu dans les données.

Note *On parle indifféremment du composant `TDataSource` ou d'une `DataSource`. Vous serez donc amené à rencontrer l'une ou l'autre de ces dénominations.*

Ajout d'un TDataSource

Le composant `TDataSource` étant maintenant sur votre fiche, vous devez le connecter. Ainsi, les composants orientés données pourront utiliser le composant `TTable` que vous avez défini. Procédez selon les étapes suivantes :

1. Ajoutez un composant `TDataSource` à votre fiche.
2. Mettez StudInfo dans la propriété `DataSet`. Cela relie le `TDataSource` au composant `TTable` Studinfo, qui accède à la table STUDINFO.DB de la base de données DBDEMOS.
3. Changez le nom du TDataSource en `dsStudent`.

Contrôles orientés données : voir et modifier les données

Vous avez à présent fourni à Delphi tous les renseignements nécessaires pour communiquer avec une table. Cependant, il est nécessaire de décider de quelle façon doivent s'afficher les données, et comment les manipuler. La façon la plus simple est d'utiliser des contrôles orientés données. La plupart de ces contrôles servent à lier un champ à un composant visuel. Par exemple, il est souvent nécessaire d'avoir une boîte d'édition contenant la valeur d'un champ d'une table. Les contrôles orientés données standard sont très puissants, simples d'utilisation, et Delphi en possède une large gamme. Excepté le DBNavigator, tous les composants de l'onglet ContrôleBD sont utilisés pour afficher les données d'une base. Pour commencer, ajoutez un composant DBGrid dans votre exemple.

Ajout d'un composant orienté données : le composant *DBGrid*

Le composant `DataSource` étant prêt, attachons-lui les contrôles orientés données :

1. Placez le contrôle `DBGrid` sur votre fiche. Comme ce composant sera visible lors de l'exécution, vous devez changez sa taille de façon à ce qu'il occupe une large portion de la fiche.
2. Mettez dsStudent dans la propriété `DataSource`. Cela relie le contrôle aux données.

3. Que s'est-il passé ? Dès que vous avez renseigné la propriété `DataSource`, les noms des champs sont automatiquement apparus dans la grille. Si des données se trouvaient dans la table STUDINFO.DB, elles se sont également affichées. Cependant, si la propriété `Active` est restée à False, rien n'est visible.
4. Lancez l'application. Vous pouvez réellement entrer des données dans la grille. Tapez un numéro de sécurité sociale, puis appuyez sur Tab, et entrez un nom d'étudiant. Parcourez les champs à l'aide de Tab jusqu'à la fin de l'enregistrement. A la fin de l'enregistrement, le contrôle DBGrid fait passer automatiquement le curseur sur la ligne suivante. Ajoutez encore quelques enregistrements. Remarquez qu'en créant deux enregistrements comportant le même numéro de sécurité sociale, une erreur se produit. Nous gérerons cela un peu plus loin.
5. Fermez l'application, et ajoutez un nouveau contrôle : le contrôle `DBEdit`. Il permet d'afficher un champ de la table. A l'opposé du contrôle grille permettant d'afficher plusieurs enregistrements, le contrôle `DBEdit` affiche un champ de l'enregistrement courant. L'enregistrement courant est celui utilisé par le moteur de bases de données à un instant donné. Le contrôle grille indique l'enregistrement courant en plaçant une flèche à sa gauche.
6. Ajoutez un contrôle `DBEdit` à votre fiche, et mettez sa propriété `DataSource` à dsStudent. Vous devez maintenant spécifier le champ à afficher. Mettez SSN dans la propriété `DataField`.
7. Lancez l'application. La Figure 11.7 montre l'application en cours d'exécution.

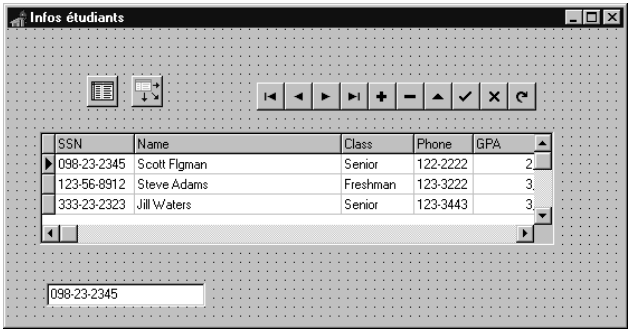


Figure 11.7
Application utilisant les composants TTable, DataSource, *et des contrôles orientés données.*

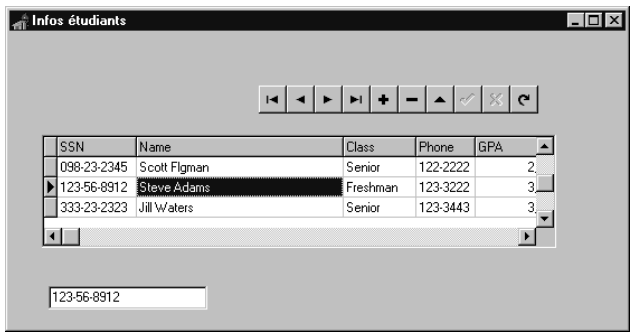
Vous remarquerez que lorsqu'un enregistrement est déplacé ou sélectionné dans la grille, cela modifie l'enregistrement courant. Cette application donne un certain contrôle à l'utilisateur, mais il serait difficile de se déplacer ainsi dans une grosse table. Delphi possède un contrôle permettant de se déplacer dans une table : le DBNavigator. C'est une barre d'outils intégrée permettant une navigation facile dans une table.

Ajout d'un *DBNavigator*

La base de données est fonctionnelle, vous devez maintenant ajouter des outils permettant de naviguer d'un enregistrement à l'autre. Vous pourriez coder ces fonctionnalités vous-même, mais il existe une méthode plus simple : l'utilisation d'un contrôle `DBNavigator`. Procédez ainsi pour l'ajouter à votre application :

1. Sélectionnez le contrôle DBNavigator et placez-le sur la fiche. Mettez sa propriété `DataSource` à dsStudent.
2. Mettez la propriété `ShowHint` à True. Cela provoque l'affichage d'une bulle d'aide chaque fois que la souris est placée sur l'une des icônes de la barre d'outils.
3. Lancez l'application. Remarquez avec quelle facilité vous pouvez vous déplacer d'un enregistrement à l'autre, aller au premier ou au dernier enregistrement, ou éditer des enregistrements. La Figure 11.8 montre la nouvelle application avec l'ajout du DBNavigator.

Figure 11.8
Ajout d'un DBNavigator.



Vous avez créé une application entièrement fonctionnelle sans écrire une seule ligne de code. Voyons maintenant comment manipuler les contrôles de base de données depuis Delphi à l'aide de Pascal Objet.

Accéder aux bases de données à l'aide de codes

Après avoir vu comment accéder aux bases de données en manipulant des composants visuels grâce à l'EDI, vous allez apprendre la manipulation des composants en utilisant la couche objet de Pascal. Les composants bases de données possèdent un impressionnant jeu d'événements, de propriétés et de méthodes pouvant être appelés depuis une unité Delphi.

Etat de DataSet

Un DataSet (ensemble de données) peut se trouver dans l'un des six états suivants : dsEdit, dsBrowse, dsInsert, dsInactive, dsSetKey ou dsCalcFields. L'état vous donne des informations sur les interactions entre l'application et les données. Pour visualiser des enregistrements, l'état devrait être dsBrowse. Si une application doit modifier un enregistrement, le DataSet doit être mis dans l'état dsEdit. La vérification de l'état d'un DataSet ne peut se faire qu'à l'exécution en consultant la propriété `State`. Pour changer cet état, vous pouvez utiliser les méthodes `Insert`, `Edit`, `Post`, `Cancel`, ou `Append`. La méthode `Insert` a pour effet d'insérer un nouvel enregistrement à la position courante et de passer cet enregistrement en mode édition. Pour éditer l'enregistrement en cours, il est possible d'utiliser la méthode `Edit`.

L'une des fonctionnalités les plus puissantes du moteur de bases de données Borland est la possibilité de valider ou d'annuler des changements pendant l'édition des enregistrements. Quand un DataSet est en mode Edit, on peut effectuer des changements dans un grand nombre de champs. Quand tous les changements sont faits, ils sont validés en appelant la méthode `Post`. Cependant, si pour une raison ou une autre l'application a besoin de restaurer les valeurs précédentes, on peut utiliser la méthode `Cancel`. Le Listing 11.1 montre comment valider ou annuler les changements opérés sur un enregistrement.

Listing 11.1 : Confirmation des modifications

```
Var
  Save:integer;

begin
   Save :=Application.MessageBox('Enregistrer ?', 'Confirmation',mb_yesno);
   if Save = IDYES then
      Studinfo.Post {Valider la transaction}
   else
       StudInfo.Cancel {Annuler la transaction};
end; {Procédure}
```

Analyse

Dans cet exemple, une boîte de dialogue est affichée dans laquelle on demande à l'utilisateur de confirmer les modifications apportées à l'enregistrement qu'il vient d'éditer. Si l'utilisateur choisit OUI, on exécute alors la méthode `Post` sur le DataSet, afin de sauvegarder les modifications. S'il choisit NON, on applique alors la méthode `Cancel` pour annuler les modifications apportées lors de la phase d'édition. Il y a un problème dans ce bout de code : si le DataSet n'est pas en mode dsEdit et qu'on tente d'exécuter une méthode `Post` ou `Cancel`, une erreur se produira. Dès lors, comment vérifier qu'on se trouve bien en mode édition ? La propriété `State` vous permet de savoir dans quel mode est le DataSet. Une version plus fiable de ce code figure dans le Listing 11.2.

Listing 11.2 : Confirmation des modifications et vérification de la propriété State

```
Var
  Save:integer;

begin
  if Studinfo.State = dsEdit then
   begin
     Save :=Application.MessageBox('Enregistrer ?,'Confirmation',mb_yesno);
     if Save = IDYES then         Studinfo.Post {Valider la transaction}
     else
         StudInfo.Cancel; {Annuler la transaction}
   end
  else
   Application.MessageBox('Mode incorrect','Erreur',mb_ok);
end; {Procédure}
```

Analyse *Ces deux exemples montrent comment contrôler les enregistrements à partir du code. Votre application peut prendre la décision logique de valider ou d'annuler les modifications. Voyons maintenant comment accéder aux champs de données.*

Accéder aux champs

Il est souvent utile pour un programme de pouvoir accéder aux champs d'une base de données et de les modifier. Delphi facilite ces manipulations. La propriété `Fields` d'un DataSet décrit plus que les valeurs des champs. Elle contient également des informations sur la structure de la table. Par exemple, la propriété `Fields` peut donner le nom des champs, leur type, leur taille, et les valeurs de l'enregistrement courant. Les objets TTable possèdent un tableau de champs. Ce tableau peut être modifié en utilisant l'éditeur de champs et en ajoutant, en enlevant ou en modifiant des définitions de champs. Pour commencer par le cas le plus simple, si l'application connaît la définition de la table, elle peut accéder aux champs directement sans se préoccuper de ce qu'ils représentent ou de leur type. Dans la table Studinfo, par exemple, SSN est la première colonne. Dès lors, `StudInfo.Fields[0].AsString` représente la valeur de SSN pour l'enregistrement courant. Remarquez que les indices de tableau commencent à zéro et que le type des données doit être connu afin d'y accéder correctement. Si vous voulez afficher le numéro de sécurité sociale dans une boîte de texte, vous pouvez écrire :

```
SSN.Text := 'NoSS='+StudInfo.Fields[0].AsString
```

Supposons que vous vous soyez trompé, et que vous ayez cru que SSN était un nombre entier. Vous auriez donc tapé :

```
SSN.Text :='NoSS='+IntToStr(StudInfo.Fields[0].AsInteger);
```

La compilation se fait sans erreur. Cependant, quand l'instruction est exécutée, une exception se produit. Comment l'éviter ? Il est possible de s'assurer du type et du nom de chaque colonne.

Cela peut être nécessaire si vous devez écrire un programme générique de manipulation de bases de données dans lequel les types sont inconnus *a priori*. Le Listing 11.3 vérifie le type des données et le nom des champs avant de tenter d'accéder aux données.

Listing 11.3 : Vérification du nom du champ et de son type

```
begin
 if not(StudInfo.Fields[0].DataType=ftString) then
  begin
   Application.MessageBox('Erreur de type (champ 0)','Erreur BD',mb_ok);
   exit;
  end; {si}
 if not(comparetext(StudInfo.Fields[0].FieldName,'SSN')=0) then
  begin
   Application.MessageBox('Nom de champ erroné (champ 0)',
                          'Erreur BD',mb_ok);
   exit;
  end; {if}
 Application.MessageBox('Champ 0: Type=ftString Nom=SSN',
                        'Information',mb_ok);
end; {procédure}
```

Analyse

Dans cet exemple, le type du champ est vérifié. Si la routine rencontre autre chose qu'un champ de type `ftString` *elle affiche un message d'erreur, et sort. De la même façon, si le nom du champ n'est pas SSN, une erreur est signalée et le programme sort. Si la procédure parvient à remplir les deux conditions, vous pouvez être certain que le nom du champ et son type sont valides.*

Il est souvent difficile d'accéder à un champ par son numéro de colonne dans la table. Delphi vous permet d'accéder aux champs par leur nom. Utilisez la méthode `FieldByName` en la faisant suivre du type de champ. Par exemple, pour accéder au champ Classe de l'enregistrement en cours et mettre sa valeur dans une boîte de liste, procédez ainsi :

```
Edit1.Text:=Studinfo.FieldByName('Classe').AsString;
```

Modifier les champs dans une Table

Par défaut, un composant `TTable` utilise tous les champs de la table à laquelle il est relié. Cependant, Delphi permet au développeur de modifier les champs utilisés, d'ajouter de nouveaux champs dérivés des anciens à partir de calculs, et de fixer les attributs de chaque champ. Pour accéder à l'éditeur de champs d'un composant `TTable`, double-cliquez sur une instance de `TTable`. Cela fait surgir une boîte de dialogue où vous pouvez ajouter, enlever ou redéfinir des champs. Cette boîte donne également accès de façon simple aux composants `TField`. Un champ est un type dérivé de `TField` qui contient en plus les informations spécifiques au type de données considéré. Par exemple, si l'un des champs est une chaîne, le champ est de type

TStringField. Tout comme les autres composants, les champs possèdent des propriétés et ont des événements. Cela permet d'avoir des propriétés différentes pour chaque champ d'une table. Si par exemple vous désirez empêcher l'utilisateur de modifier un champ donné, tout en lui laissant libre accès aux autres, vous mettez la propriété ReadOnly du champ susmentionné à True. Il existe un grand nombre de propriétés relatives à l'affichage des données dans les contrôles orientés données. La propriété Alignement, par exemple, permet de spécifier si on doit aligner à droite, à gauche ou au centre, les données dans la zone d'affichage. Pour les nombres flottants, Precision permet de spécifier à quelle décimale se fait l'arrondi.

Navigation dans les enregistrements

Vous avez déjà vu le composant DBNavigator permettant de se déplacer d'un enregistrement à l'autre, ainsi que de se placer en début ou en fin de table. Cela peut également être fait depuis le code. Pour changer l'enregistrement actif, cinq méthodes sont disponibles dans le composant TTable. Elles sont décrites dans le Tableau 11.8.

Tableau 11.8 : Méthodes de navigation de DataSet

Méthode	Effet
Next	Se déplace sur l'enregistrement suivant.
Prior	Se déplace sur l'enregistrement précédent.
First	Va au premier enregistrement.
Last	Va au dernier enregistrement.
MoveBy(I)	Se déplacer de I enregistrements. Si I est positif, vers la fin de la table, si I est négatif, vers le début.

Il est important de savoir qu'en mode Edition, un changement d'enregistrement actif provoque un Post implicite, et que les modifications sont donc enregistrées. Il est alors trop tard pour effectuer un Cancel qui annulerait les modifications. Lorsqu'on navigue dans une table, il est souvent utile de savoir si l'on est arrivé au premier ou au dernier enregistrement. Pour cela, on utilise les propriétés BOF et EOF.

BOF est vraie lors du lancement de l'application, après un appel à la méthode First, ou quand une méthode Prior échoue car l'enregistrement est déjà le premier de la table.

EOF est vraie si la table est vide, après un appel à la méthode Last, ou si un appel à la méthode Next échoue parce que l'on est en fin de l'ensemble de données.

L'une des actions les plus classiques sur un dataset est d'effectuer une opération sur chaque enregistrement. Dans l'exemple suivant, la procédure se place sur le premier enregistrement et boucle sur chacun des enregistrements suivants jusqu'à ce que la propriété EOF soit vraie. Sur chaque enregistrement on exécute la méthode Edit afin de pouvoir modifier la valeur de l'enregistrement. Dans cet exemple, un numéro de chambre de dortoir est assigné de façon aléatoire

à chaque étudiant Pour cela, on génère un nombre aléatoire entre 0 et 1. Si le nombre est plus petit que 0,5 on affecte l'étudiant à un des dortoirs (`StudInfo.Fields[6]`), sinon à l'autre. Après avoir effectué les modifications, on exécute la méthode `Post` pour sauvegarder les modifications. Le code source de cet exemple est dans le Listing 11.4.

Listing 11.4 : Déplacement dans un DataSet pour modifier chaque enregistrement

```
procedure TForm1.AssnDormsClick(Sender: TObject);
begin
  Studinfo.First; {Premier enregistrement}
  StudInfo.DisableControls; { accélère le traitement }
  while Not(Studinfo.EOF) do {Dernier enregistrement ?}
    begin
     StudInfo.Edit;
     {Génération d'un nombre aléatoire}
     {Accès aux champs par la propriété Fields}
     if random > 0.5 then
       StudInfo.Fields[6].AsString := 'DortoirA'
      else
       StudInfo.Fields[6].AsString := 'DortoirB';
     StudInfo.Post; { Post explicite (inutile mais plus clair)}
     StudInfo.Next; { Enregistrement suivant }
    end; {fin de la boucle}
    StudInfo.EnableControls; { Améliore les performances }
end; {fin de la procédure}
```

Analyse *Remarquez que les contrôles sont désactivés pendant la boucle afin d'améliorer les performances, puis rétablis à la fin de la boucle. De cette façon, les contrôles ne se remettent pas à jour pendant les opérations, ce qui occasionnerait une perte de temps.*

Champs calculés

Vous souhaiterez souvent extraire des données à partir d'une base de données, sans qu'il soit logique de stocker les informations calculées dans la table. Par exemple, si un inventaire est constitué d'une table où chaque item voit figurer son nombre en stock et son poids, il est inutile de stocker le poids total des items en question. Ce poids total peut être inféré en multipliant le nombre d'items par le poids de chacun. Si vous avez 120 lecteurs de CD-ROM pesant 2 kg chacun, le poids total est de 240 kg (cela peut être utile pour calculer des frais de transport). Il faudrait donc que Delphi soit à même de calculer et d'afficher le poids total chaque fois que c'est nécessaire. Stocker ce poids total dans la table est compliqué, puisqu'il faut le recalculer à chaque modification, et encombrant, parce que ces données sont redondantes. Il existe donc un type de champ adapté à ce genre de manipulations, les *champs calculés*. On les appelle champs calculés parce qu'ils se calculent à partir des autres champs de la table. L'avantage est qu'ils

ne sont pas stockés dans la table : ils sont réévalués chaque fois qu'il est nécessaire de les consulter. Pour ajouter des champs calculés, utilisez l'éditeur de champs auquel on accède par un double-clic sur le composant `TTable`. Cliquez sur le bouton Définir. Vous devrez fournir un nom pour le nouveau champ et préciser son type. Un nom de composant est créé par défaut, mais vous pouvez le modifier à votre guise. L'étape suivante consiste à insérer le code permettant de calculer la valeur du champ. Delphi exécute la procédure dans le Gestionnaire d'événements `OnCalcFields` chaque fois qu'il est nécessaire de recalculer un champ. Tant que le `DataSet` est dans le Gestionnaire d'événements `OnCalcFields`, sa propriété `State` vaut `dsCalcFields`. Dans cet état, seuls les champs calculés peuvent être modifiés.

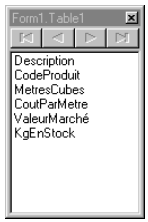
Regardons une application simple. La boutique du Jeune Campeur veut une base de données. Après analyse, on arrive à la table détaillée au Tableau 11.9.

Tableau 11.9 : Attributs de la table du Jeune Campeur

Nom	Description
`CodeProduit`	Index du produit
`MetresCube`	Nombre de mètres en stock
`CoutParMetre`	Prix au mètre
`KgParMetre`	Poids au mètre
`ValeurMarchande`	Valeur totale du stock
`KgsEnStock`	Poids total en stock

On pourrait créer cette table telle quelle et créer une application pour accéder aux données et les mettre à jour. Cependant, il existe des relations entre les champs. Quand le champ MetresCubes change, le poids total du stock et la valeur totale changent. D'autre part, si les prix changent, la valeur totale change également. Dès lors, créer la table en stockant tous ces champs demanderait des mises à jour complexes chaque fois qu'une valeur est modifiée. En utilisant des champs calculés, on élimine une bonne partie du travail. Nous allons donc construire la table en omettant les champs ValeurMarchande et KgsEnStock dans la structure. Ajoutez ces deux champs sous forme de champs calculés. La Figure 11.9 montre l'éditeur de champs utilisé pour créer le nouveau champ ValeurMarchande, de type CurrencyField.

Figure 11.9
L'éditeur de champs.



L'étape finale consiste à placer dans `OnCalcFields` le code nécessaire au recalcul des champs calculés. Cette procédure est détaillée dans le Listing 11.5.

Listing 11.5 : Code utilisé pour calculer les champs

```
procedure TForm1.InventoryCalcFields(DataSet: TDataset);

Var
   CurrentPrice : double; { conversion pour empêcher les débordements}
   LbsPerYard   : double;
   YardsInStock : double;

begin
    {stockage des champs dans des variables locales }
    {pour rendre la source plus lisible}
   CurrentPrice := Inventory.FieldByName('CoutParMetre').AsFloat;
   LbsPerYard   := Inventory.FieldByName('KgParMetre').AsFloat;
   YardsInStock := Inventory.FieldByName('MetresCubes').AsFloat;

   {Affectation des champs calculés}
   Inventory.FieldByName('ValeurMarchande').AsFloat :=
      CurrentPrice * YardsInStock;
   Inventory.FieldByName('KgsEnStock').AsFloat :=
      LbsPerYard   * YardsInStock;
end; {Procedure}
```

Analyse *Dans ce Gestionnaire d'événements, les variables locales sont initialisées à leur valeur dans l'enregistrement courant. Elles servent ensuite à calculer la valeur des champs calculés.*

Index

Un index est une façon de trier les enregistrements d'une table en fonction d'un critère, afin d'accélérer les recherches. Au sens le plus général des bases de données relationnelles, un champ peut être une clé sans pour autant être indexé. Cependant, sous Paradox et dBASE, définir une clé crée automatiquement l'index associé. Le gros avantage de cet index est qu'il accélère les tris et les recherches sur le champ indexé. Considérons par exemple une table contenant 10 000 enregistrements. Pour trouver l'enregistrement dont le Nom de Famille vaut Dupond, il faudra au pire 10 000 itérations, et en moyenne 5 000. Si par contre le champ Nom de Famille est indexé, la recherche se fera en 14 itérations en moyenne. Le gain de temps est considérable.

La plupart des SGBD permettent d'avoir plus d'un index par table. Avant de vous montrer comment créer des index supplémentaires en utilisant le Module base de données, regardons quels inconvénients cela peut avoir. Il est vrai qu'en créant un index on gagne énormément en

vitesse, mais uniquement en recherche et en tri. A l'inverse, créer un index ralentit les opérations d'insertion, de mise à jour, et fait augmenter la taille de la base. En conclusion, lorsque vous créez une nouvelle table, cherchez les colonnes qui serviront souvent à des recherches ou a des tris, et ne mettez d'index que sur celles-ci. Un des avantages de la séparation entre le système de gestion de base de données et l'application est que s'il s'avère nécessaire de créer un nouvel index pour augmenter les performances, cela peut être fait à n'importe quel moment.

Nous avons appris qu'un index est automatiquement créé pour la clé. Est-il nécessaire pour en construire un de disposer d'un champ-clé ? Non, seul le premier index doit être sur un champ clé, les autres n'ont pas cette restriction. La façon la plus simple de créer des index sur autre chose que la clé primarité est d'utiliser le Module base de données. Voyons comment faire sur STUDINFO.DB.

Ajout d'un index

Maintenant que vous connaissez la fonction des index, ajoutons-en un à notre base de données :

1. Ouvrez le Module base de données en utilisant l'option Outils de la barre de menu.
2. Dans le Module base de données, choisissez Fichier/Ouvrir/Table.
3. Dans le lecteur (ou Alias), choisissez DBDEMOS.
4. Choisissez STUDINFO.DB, puis cliquez sur OK.

Note *si Delphi est lancé et que l'application accède à la table, mettez à False la propriété* `Active` *de l'objet TTable, sans quoi il vous sera impossible de modifier la table.*

5. Choisissez Table/Restructurer. Ceci vous donne accès à tous les champs de la table STUDINFO.
6. Dans la liste déroulante Propriétés de la table, choisissez Index Secondaires.
7. Cliquez sur Définir.
8. Choisissez le champ Nom et cliquez sur le bouton comportant une flèche vers la droite. Cette opération déplace le champ Nom dans la boîte des champs indexés. Cliquez sur OK.
9. Le Module base de données vous demande un nom pour l'index. Tapez NomInd, puis cliquez sur OK. La Figure 11.10 montre le nouvel index secondaire ajouté à la table.
10. Cliquez maintenant sur Enregistrer, puis sur Fichier/Quitter : vous quittez le Module base de données.

Figure 11.10
Définition d'un index secondaire sur un champ.

Maintenant que vous savez indexer une table, comment faire les recherches sur un enregistrement donné ?

Trier des enregistrements

Quand on navigue d'un enregistrement à l'autre, l'ordre de parcours est déterminé par l'index actif. La propriété `IndexName` de l'objet TTable détermine l'index en cours d'utilisation. Si aucun index n'est spécifié, l'index primaire est utilisé pour trier les enregistrements. L'autre façon d'indiquer à Delphi l'index à utiliser est la propriété `IndexFieldNames`. Remarquez que ces deux propriétés ne peuvent être utilisées en même temps. Chaque fois que vous modifiez l'une d'entre elles, l'autre s'efface. Prenez l'exemple du cas où vous désirez changer l'ordre de tri en passant par un menu. Vous pouvez alors changer la propriété `IndexFieldNames` (voir Listing 11.6).

Listing 11.6 : Changement de l'ordre de tri par un menu

```
procedure TForm1.Order_By_NameClick(Sender: TObject);
begin
  Order_By_Name.Checked := True;
  Order_By_SSN.Checked := False;
  StudInfo.IndexFieldNames := 'Nom';
end;
```

Analyse *Au départ, l'élément de menu Order_By_SSN est non coché. On le coche afin d'indiquer quel index est utilisé, puis on modifie la propriété* `IndexFieldNames` *de façon concordante.*

Rechercher des enregistrements

Jusqu'à présent nous nous sommes contentés de naviguer dans la table et de modifier l'ordre de tri des enregistrements. Supposons cependant que vous désiriez accéder à un enregistrement

particulier de la table. L'une des méthodes possibles consiste à explorer la table enregistrement par enregistrement jusqu'à trouver le bon. Cette méthode est envisageable mais elle néglige grandement les possibilités du moteur de bases de données. Les outils inclus dans l'objet TTable permettent de faire des recherches de façon rapide et simple. Il y a cependant une limitation importante à ne pas oublier : seuls les champs indexés peuvent être utilisés avec ces outils.

Deux techniques sont disponibles pour effectuer des recherches : `SetKey` et `FindKey`. La démarche utilisée pour rechercher des enregistrements n'est pas intuitive. Il faut tout d'abord mettre la table dans l'état approprié. Pour cela, on utilise la méthode `SetKey`. Une fois cette méthode appelée, la table est dans l'état `dsSetKey`. Par exemple, pour passer la table Studinfo dans l'état `dsSetKey`, il faudra exécuter :

```
Studinfo.SetKey;
```

Une fois que la table est dans l'état approprié, assignez au champ sur lequel vous effectuez la recherche la valeur désirée. Cette approche est un peu surprenante, parce que votre code semble affecter la valeur recherchée au champ au lieu de chercher le champ contenant la valeur recherchée. Mais n'oubliez pas que vous n'êtes pas dans l'état dsEdit, mais dans l'état dsSetKey. Cette instruction a pour effet de spécifier au moteur de base de données la valeur recherchée. Commençons par un exemple simple, une recherche sur la clé primaire. Pour trouver l'enregistrement correspondant à la clé, donnez au champ clé la valeur recherchée. Par exemple, pour rechercher dans STUDINFO l'étudiant dont le numéro de sécurité sociale est 1661193105687, vous devez exécuter :

```
Studinfo.FieldByName('SSN'). Assting :='1661193105687';
```

Enfin, pour se placer sur l'enregistrement recherché, vous devez terminer la recherche en exécutant la méthode `GotoKey`. Dans notre cas, vous devrez faire :

```
StudInfo.GotoKey;
```

Qu'arrive-t-il si l'enregistrement recherché n'existe pas ? L'enregistrement courant ne changera pas, et GotoKey renverra False.

Cette méthode est efficace si vous recherchez une valeur exacte. Cependant, comment faire si seule une partie du champ à rechercher est connue ? Dans ce cas, on utilise la méthode `GotoNearest` qui recherche l'élément ressemblant le plus au critère de recherche. Supposons que vous désiriez implémenter une fonction recherchant un étudiant connaissant uniquement le début de son numéro de sécurité sociale. Le Listing 11.7 propose une solution à ce problème.

Listing 11.7 : Recherche sur un numéro de sécurité sociale

```
procedure TForm1.SearchSSNClick(Sender: TObject);
begin
  StudInfo.SetKey;
  Studinfo.FieldByName('SSN').AsString := teSSNSearch.text;
  Studinfo.GotoNearest;
end;
```

Analyse

Cette procédure place la table StudInfo dans l'état SetKey, puis assigne au champ SSN la valeur de la boîte de texte teSSNSearch. Lors de l'exécution de `GotoNearest`*, l'enregistrement courant devient celui dont le numéro de sécurité sociale est le plus proche de celui recherché.*

Naturellement, tout cela semble un peu long et compliqué. Il existe heureusement des raccourcis pour faire toutes ces opérations en une seule ligne : les méthodes `FindKey` et `FindNearest`. Lors de leur exécution, Delphi encapsule les instructions `SetKey`, `Search Criteria`, et `Goto` dans une seule fonction à laquelle on passe les valeurs à rechercher. Le Listing 11.7 peut être simplifié pour obtenir le Listing 11.8.

Listing 11.8 : Recherche avec moins de code (plus difficile à lire)

```
procedure TForm1.SearchSSNClick(Sender: TObject);
begin
    Studinfo.FindNearest([teSSNSearch.text]);
end;
```

Analyse

Un des avantages de la méthode `SetKey` *est que vous n'êtes pas obligé de connaître l'ordre des champs indexés. Si vous disposez par exemple d'une table indexée sur Nom et Prénom, vous pouvez combiner les méthodes* `SetKey` *et* `FieldByName` *pour affecter une valeur au champ Nom, et une autre au champ Prénom. Vous utilisez ensuite* `GotoKey` *ou* `GotoNearest` *et n'avez jamais de problèmes. Par contre, avec la méthode* `FindKey`*, vous devez fournir les deux valeurs en une seule fois. Mais, faut-il appeler*

```
IRS.FindKey(['Anne-Sophie','Juppe']);
```

ou plutôt

```
IRS.FindKey(['Juppe','Anne-Sophie']);
```

Le problème est qu'à moins d'être sûr de l'ordre dans lequel Delphi attend les arguments, vous risquez de ne pas trouver les enregistrements que vous recherchez.

Recherche sur des index secondaires

Effectuer une recherche sur autre chose que l'index primaire se fait en modifiant la valeur de la propriété `IndexName` ou `IndexFieldNames` tout comme vous l'aviez fait pour modifier l'ordre de tri. En conséquence, pour faire une recherche par nom dans la table Studinfo vous exécutez le code du Listing 11.9.

Listing 11.9 : Recherche sur une clé secondaire

```
procedure TForm1.SearchNameClick(Sender: TObject);
begin
  StudInfo.SetKey;
  StudInfo.IndexName := 'NameInd';
```

```
  Studinfo.FieldByName('Nom').AsString := teNameSearch.text;
  Studinfo.GotoNearest;
end;
```

Analyse *Ce programme effectue la même recherche que celui du Listing 11.7, mais nous avons spécifié un index de recherche autre que l'index primaire, grâce à la ligne*

```
StudInfo.IndexName := 'NameInd';
```

Se limiter à une étendue d'enregistrements

Il est souvent intéressant de restreindre les enregistrements d'une table selon un certain critère. Par exemple, vous pouvez décider de ne considérer dans la table Studinfo que les étudiants en première année, ou bien uniquement ceux dont la moyenne dépasse 3,6. Ces opérations limitent l'étendue de la table car vous travaillez sur un sous-ensemble des données. Une fois que vous avez défini l'étendue, tout se passe comme si les enregistrements n'en faisant pas partie n'existaient pas. La méthode `SetRange` permet de définir une étendue. Le Listing 11.10 vous montre comment choisir un index par son nom et délimiter une étendue. La dernière procédure montre comment supprimer une contrainte d'étendue en utilisant la méthode `CancelRange`.

Listing 11.10 : Affichage d'un sous-ensemble de table grâce à la méthode SetRange

```
procedure TForm1.RangeFreshClick(Sender: TObject);
begin
  Studinfo.IndexName := 'ClassInd';
  StudInfo.SetRange(['Deug1'],['Deug1']);
end;

procedure TForm1.RangeDeansClick(Sender: TObject);
begin
  Studinfo.IndexName := 'GpaInd';
  StudInfo.SetRange([3,6],[5]);
end;

procedure TForm1.All1Click(Sender: TObject);
begin
  StudInfo.CancelRange;
end;
```

Analyse *Ce programme illustre plusieurs types d'utilisation de la méthode `SetRange` pour limiter la quantité d'informations renvoyée. La première procédure spécifie l'index puis l'étendue. La seconde utilise l'index `GPAInd` avant de définir l'étendue entre `3.6` et `5`. Enfin, la dernière*

procédure effectue un `CancelRange` pour permettre la recherche dans tous les enregistrements.

Validation des saisies dans une Table

Dans un monde meilleur, les utilisateurs se comporteraient comme l'imaginent les programmeurs. Ce n'est pas le cas, un utilisateur finit toujours par trouver une façon bien à lui d'utiliser une application. Bien qu'il soit impossible de tout prévoir, il existe cependant quelques techniques pour stabiliser l'application et forcer l'utilisateur malicieux à saisir des données conformément à l'esprit général du développeur. Nous allons donc voir les méthodes utilisables pour contrôler les données saisies dans une base de données :

- Composants de sélection
- Masques de saisie
- Contraintes placées au niveau de la base de données
- Propriété `Validation` des objets `Tfield` et `TTable`.

Composants de sélection au lieu des composants à saisie libre

Pour que votre application soit conviviale, ne laissez pas l'utilisateur commettre d'erreurs : utilisez des boutons radio, des listes déroulantes, etc., chaque fois que l'occasion se présente. Les contrôles orientés données les plus dangereux sont les boîtes de texte et les champs mémo. Ils permettent à l'utilisateur de rentrer n'importe quoi. En utilisant cette technique, vous dissiperez quelques causes de conflits avec les utilisateurs.

Masques de saisie

La première chose possible à faire pour inviter l'utilisateur à saisir ce qu'on lui demande consiste à utiliser des masques de saisie. Un masque de saisie est une propriété de chaque composant à zone de saisie, empêchant les saisies qui ne suivent pas un format prédéfini. Si vous désirez que l'utilisateur saisisse un numéro de téléphone, certains tenteront d'écrire (16 1) 44 44 44 44, d'autres 16144444444. Ces deux saisies sont valides, mais il serait bon qu'elles soient rentrées avec le même format. On construit un masque de saisie en spécifiant des règles contraignant chaque valeur de la zone de saisie. Par exemple, un numéro de téléphone pourrait avoir comme masque 00\-00\-00\-00;1;_, qui permettrait de saisir 44-44-44-44, mais pas A3-45-11-42, car le masque spécifie que tous les caractères doivent être des chiffres, et qu'ils doivent être saisis sous la forme XX-XX-XX-XX. Merveilleux, mais comment définir un masque de saisie ? Delphi comporte une boîte de dialogue dédiée aux masques de saisie. Pour faire surgir cette boîte, double-cliquez sur la propriété MaskEdit d'un objet Tfield (voir Figure 11.10). L'éditeur de masques comprend une liste des formats couramment utilisés ainsi qu'un champ

pour créer ses propres masques. Pour en savoir plus sur MaskEdit, consultez la rubrique d'aide de Delphi.

Figure 11.11
L'éditeur de masque de saisie.

Contraintes placées au niveau de la base de données

La propriété `MaskEdit` est efficace au niveau de l'application. Cependant, si différentes applications accèdent à la même base de données, il serait plus agréable de pouvoir gérer les contraintes au niveau de la base et non au niveau de l'application. Chaque SGBD a ses propres fonctionnalités, mais la plupart possèdent des outils pour contraindre les données d'un champ. Pour plus de simplicité, nous allons nous restreindre aux critères de validation de Paradox. Ces critères sont stockés dans la base de données, ce qui garantit son intégrité même si plusieurs applications y accèdent. Les contraintes disponibles sont :

- Rendre obligatoire la saisie d'un champ.
- Fixer une valeur minimale pour un champ.
- Fixer une valeur maximale pour un champ.
- Donner une valeur par défaut à un champ.

On utilise le Module base de données pour définir ces critères. N'oubliez pas que si votre application utilise la base de données que vous allez manipuler avec le Module base de données, vous devez mettre la propriété `Active` à False avant de lancer le Module base de données, ou bien fermer le projet. Pour ajouter des critères de validation, procédez ainsi :

1. Sélectionnez Outils/Module base de données pour lancer le Module base de données.
2. Choisissez Fichier/Ouvrir/Table, et sélectionnez la table à laquelle vous désirez ajouter des critères de validation.
3. Choisissez Table/Restructurer. Cela fait apparaître la liste des champs de la table, avec leur type.
4. Vérifiez que la liste déroulante Propriétés de la table contient Intégrité référentielle Contrôles de validité.

5. Sélectionnez le champ que vous désirez contraindre et saisissez les contraintes dans la boîte de dialogue. Par exemple, pour mettre une contrainte sur le champ Moyenne de la table Studinfo, mettez 0 dans la boîte Minimum et 4,0 dans la boîte Maximum.

Note *Le développeur n'est pas toujours propriétaire de la base de données. Il arrivera parfois qu'une application Delphi viole les contraintes de validité d'une base de données. Dans ce cas, une erreur se produira.*

Méthode Cancel

La méthode `Cancel` peut être utilisée en dernier recours contre les erreurs de saisie de l'utilisateur. Dans certains cas, les contraintes de validation sont trop sophistiquées pour être formulées en termes de masques de saisie ou de contraintes au niveau de la base. Par exemple, le professeur Nimbus voudrait créer une base de données pour stocker ses nombres premiers favoris. La contrainte de validité est donc quelque peu complexe à vérifier. La solution consiste à utiliser un objet dbTextEdit pour demander à l'utilisateur ses nombres premiers favoris, un dbNavigator pour se promener dans la base et un bouton Enregistrer pour sauvegarder les modifications. Le problème principal réside dans la validation des données. Une possibilité consisterait à passer dans l'état Edit chaque fois que l'utilisateur entre dans la zone de saisie, et à désactiver la barre de navigation pour le forcer à utiliser le bouton Enregistrer en fin de saisie. On placerait alors le code de validation dans le Gestionnaire d'événements `OnClick`

Note *Rappelons qu'un nombre premier n'a d'autres diviseurs que 1 et lui-même. 3 est premier, mais 14 ne l'est pas car il est divisible par 1,2,7, et 14). L'algorithme utilisé ici consiste à tenter de diviser le nombre par tous les entiers compris entre 2 et n-1, et à refuser la validation si le reste de la division est 0. Il y a nettement plus efficace.*

Le Listing 11.11 montre comment utiliser la méthode `Cancel` pour confirmer qu'un nombre est premier.

Listing 11.11 : Validation des nombres premiers par la méthode Cancel

```
procedure TForm1.EntPrimeEnter(Sender: TObject);
begin
  Nav.Enabled := False;  {Désactiver le Navigator}
  Primes.Edit;           {Passage en mode edit}
end;

procedure TForm1.SaveChangesClick(Sender: TObject);

Var
 IsPrime : Boolean;
 Count : Integer;
 NumToTest : Integer;
```

```
begin
   IsPrime := True;
   Count := 2;
   NumToTest := StrToInt(EntPrime.Text);
   While (Count < NumToTest) and (IsPrime) do
     begin
       if NumToTest mod Count = 0
         then IsPrime := False; { un diviseur et c'est fini!}
       inc(Count); { incrémentation de count}
     end;
   {Si le nombre est premier, stockons-le}
   {Sinon, utilisons la méthode Cancel}
   if (IsPrime) then
      Primes.Post
     else
      begin
        Application.MessageBox('Le nombre n''est pas
premier!','Attention',MB_OK);
        Primes.Cancel;
      end;
   Nav.enabled := true; {Réactivons le Navigator}
end; {Procédure}
```

Analyse *Ce programme vérifie qu'un nombre est premier avant de l'envoyer dans la base de données. S'il n'est pas premier, on appelle la méthode* `Primes.Cancel` *pour annuler la mise à jour sur la base de données. Si le nombre est premier, on appelle la méthode* `Primes.Post` *pour confirmer que la donnée doit être ajoutée à la base.*

Propriétés de validation des objets Tfield et TTable

Nous avons vu trois techniques pour assurer la cohérence des données saisies par l'utilisateur : vérifier ce qu'il saisit, valider au niveau de la base, et forcer l'utilisateur à passer par un point de contrôle. La dernière technique que nous allons aborder concerne l'utilisation d'événements et d'exceptions. En utilisant les Gestionnaires d'événements des objets table et champ, une application peut valider les données avant qu'elles ne soient enregistrées. Cette méthode laisse plus de souplesse à l'utilisateur quant à l'ordre de saisie et d'édition des champs. Pourquoi exécuter les validations au niveau de la table et des champs ? Dans certains cas, un champ peut être déclaré valide sans connaître d'informations sur le reste de la base. C'était par exemple le cas pour les nombres premiers. Imaginons cependant une application de caisse enregistreuse. Une entrée n'est valide que si la somme encaissée est supérieure à la somme rendue, et il n'est donc pas possible d'effectuer de validation similaire à l'exemple précédent.

Pour effectuer un contrôle de validation au niveau des champs, il faut mettre en place un Gestionnaire d'événements sur l'événement `OnValidate`. Cet événement a lieu juste avant l'écriture des données dans la base. Si vous voulez que les données soient enregistrées, laissez la procédure s'exécuter. Sinon, mettez en place une exception afin de bloquer l'exécution de la procédure. Le Listing 11.12 donne une autre version du problème des nombres premiers.

Listing 11.12 : Validation par l'événement OnValidate

```
type
   EInvalidPrime = class(Exception);

{...}
procedure TForm1.PrimesFavPrimesValidate(Sender: TField);

Var
 IsPrime : Boolean;
 Count : Integer;
 NumToTest : Integer;

begin
 try
   IsPrime := True;
   Count := 2;
   NumToTest := StrToInt(EntPrime.Text);
   While (Count < NumToTest) and (IsPrime) do
     begin
       if NumToTest mod Count = 0
         then IsPrime := False;
       inc(Count);
     end; {while}
   if not(IsPrime) then
      begin
        raise EInvalidPrime.Create('Le nombre n''est pas premier');
      end; {else}
  except du
   on EInvalidPrime do
     begin
       Application.Messagebox('Nombre non premier','Erreur',MB_OK);
       EntPrime.Text :='';
       if (Primes.State = dsInsert) or (primes.State=dsEdit)
        then Primes.cancel;
     end;
```

```
  end;
end;
```

Analyse *Remarquez les clauses* `Try` *et* `Except` *utilisées pour intercepter l'exception lorsque vous vous apercevez qu'un nombre n'est pas premier, ainsi que la méthode* `Exception.Create` *utilisée pour créer une exception utilisateur. Pour effectuer la validation au niveau de l'enregistrement, la technique serait semblable, mais appliquée à l'événement* `BeforePost`.

Gestion des exceptions

Il est manifeste que votre application doit être sensible aux données non valides. Mais que faire lorsque le cas se produit ? Il y a des cas où des exceptions se produisent alors que les contrôles de validité n'ont rien signalé d'anormal. Voici quelques-unes des situations les plus courantes :

- **Clés dupliquées** — Même si un champ est rempli de façon cohérente, il faut s'assurer que la clé n'est pas dupliquée. Si vous tentez par exemple d'insérer un nouvel étudiant et que le numéro de sécurité sociale que vous avez saisi appartient à un autre étudiant, votre saisie est erronée. Si Delphi laissait les deux enregistrements apparaître dans la base, le champ SSN ne serait plus une clé, ce qui rendrait la base incohérente.
- **Erreur de validation de la base** — Nous avons vu précédemment qu'il était possible de mettre des contrôles de validation au niveau de la base de données. Que se passe-t-il si l'une de ces contraintes est violée ? Une exception se produit. Votre application doit être en mesure de la détecter et de réagir avec élégance.
- **Opération invalide dans l'état actuel** — Il est dangereux de supposer que la table est dans un certain état avant de faire des manipulations dessus, surtout dans un contexte de gestion événementielle. Le comportement de la table varie selon son état (dsEdit, dsBrowse, dsInsert, dsInactive, dsSetKey, dsCalcFields). Si par exemple la table est dans l'état dsEdit, l'instruction

  ```
  Studinfo.FieldByName('Classe') .AsString:='Deug1';
  ```

  Mettra Deug1 dans le champ Classe, alors que la même instruction effectuée dans l'état dsSetKey indiquerait simplement que l'on recherche le premier Deug1. Ces erreurs sont extrêmement difficiles à retrouver car elles ne provoquent pas d'exceptions. Certaines le font quand même, par exemple l'utilisation des méthodes `Post` ou `Cancel` en mode dsBrowse.

ODBC : une couche d'abstraction supplémentaire

Le moteur de bases de données Borland peut communiquer avec un grand nombre de systèmes de bases de données. Cette caractéristique est intéressante si le développeur veut changer de système de gestion de bases de données. Pour communiquer avec n'importe quel type de système de gestion de bases de données, il existe ODBC.

Nouveau

ODBC signifie Open Database Connectivity. ODBC vous permet de communiquer virtuellement avec n'importe quel SGBD (Système de Gestion de Bases de données) grâce à une interface générique appelée pilote ODBC. Le pilote ODBC assure l'interfaçage avec un SGBD particulier en normalisant les accès à travers un ensemble d'appels à une API.

L'utilisation d'ODBC garantit à l'application un maximum de souplesse, puisqu'il suffit en théorie de changer de pilote ODBC pour changer de SGBD. En réalité, il y a quelques différences d'un SGBD à l'autre. Si la vitesse d'exécution pose problème, il est possible d'accéder directement à l'API ODBC. Cependant, cela introduit un nouveau niveau de difficulté.

Dans quelles circonstances utiliser ODBC ?

Il arrive souvent que le développeur ne sache pas à quelle échelle développer sa base (locale, à fichiers partagés, ou Client/Serveur). ODBC permet à ce développeur de créer une base de données locale, et de la transformer selon le modèle Client/Serveur avec un minimum de modifications. Il pourra également arriver que le développeur soit obligé de tenir compte d'impératifs de compatibilité. Si par exemple une application de paie utilise SyBase SQL, Delphi pourra utiliser ODBC pour se connecter à la base de données.

Dans la terminologie ODBC, un pilote ODBC est une libraire qui sait communiquer avec le SGBD sous-jacent. Les pilotes ODBC sont souvent fournis par le vendeur du SGBD. Une source de données ODBC est une instance d'ODBC utilisant un pilote particulier. Par exemple, vous pouvez disposer d'un pilote ODBC Sybase SQL server, et RHDATA est une source de données ODBC qui est un serveur SQL des données relatives aux ressources humaines. Il peut y avoir plusieurs sources de données utilisant le même pilote, par exemple INVENT pointerait vers les informations d'inventaire.

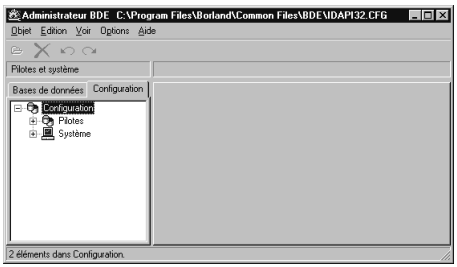
L'utilitaire ODBCAD32 sert à créer et configurer des sources de données. Pour chaque source de données les informations sont distinctes. Par exemple, une source de données ODBC Microsoft Access doit connaître l'emplacement du fichier .mdb, alors qu'une source de données SQL devra comporter l'adresse du serveur et l'interface réseau nécessaire pour y accéder.

Pour utiliser un pilote ODBC et communiquer grâce à ODBC sous Delphi, vous devez créer un nouveau moteur de bases de données. Cela peut sembler étrange, mais vous devez créer un pilote ODBC vers le SGBD voulu, et un moteur de bases de données Borland vers le pilote ODBC. Pour créer un moteur de bases de données Borland, procédez ainsi :

1. Lancez l'utilitaire de configuration de bases de données Borland depuis le menu démarrer de Windows 95 ou NT.
2. Cliquez sur l'onglet Configuration. Sélectionnez Configuration|Pilotes|ODBC. Faites un clic droit sur ODBC, et sélectionnez Nouveau dans le menu contextuel.
3. Mettez le nom du pilote SQL Link à ODBC_XXXX, où XXXX est le nom du pilote Borland que vous voulez créer (voir Figure 11.12).
4. Sélectionnez le pilote ODBC à utiliser pour accéder aux données ainsi qu'une source de données afin de créer un alias ODBC, puis cliquez sur OK. Sélectionnez Objet|Quitter pour fermer l'Administrateur BDE.

Voilà tout pour la création du pilote. Pour accéder aux données, le meilleur moyen est de créer un alias en utilisant le Module base de données.

Figure 11.12
L'Administrateur BDE.



Base de données Microsoft Access avec ODBC

Définissez la source de données ODBC de la façon suivante :

1. Lancez ODBCAD32.
2. Cliquez sur Ajouter dans la boîte de dialogue Sources de données. La liste des pilotes ODBC installés s'affiche.
3. Choisissez *.mdb puis cliquez sur Terminer.
4. Donnez un nom et une description à la source de données Access. Dans cet exemple, nous mettrons `TestAccess`.
5. Cliquez sur Choisir. Vous voyez apparaître une liste de fichiers .mdb.
6. Choisissez le fichier qui vous intéresse.
7. Cliquez sur OK. Cliquez de nouveau sur OK pour sortir du programme ODBCAD32.

Avant de pouvoir utiliser le pilote ODBC que vous venez de définir, vous devez exécuter l'Administrateur BDE (il fait partie des programmes installés dans le menu Démarrer). La nouvelle source ODBC sera alors automatiquement prise en compte. Quittez l'Administrateur pour enregistrer les modifications.

Note

Pour que la prise en compte des nouveaux pilotes ODBC se fasse automatiquement, il faut que le paramètre AUTO ODBC de l'Administrateur BDE soit à TRUE. Vous pouvez vérifier ce paramétrage de la façon suivante :

1. Exécutez l'Administrateur BDE.

2. Sélectionnez l'onglet Configuration.

3. Sélectionnez Configuration\Système\INIT.

4. La Définition de INIT apparaît alors dans le volet de droite. La propriété AUTO ODBC est sur la première ligne.

Pour utiliser l'alias ODBC sous Delphi, procédez comme ainsi :

1. Créez un nouveau projet, et ajoutez un composant TTable.
2. Mettez la propriété `DatabaseName` à `TstAccess`.
3. Si une boîte de dialogue demandant un nom d'utilisateur et un mot de passe apparaît, cliquez sur OK.
4. Notez que les tables de la base de données Access sont maintenant disponibles dans la propriété `TableName`. Choisissez une table, et continuez comme avec n'importe quelle table Delphi.

Note

Si, lorsque vous mettez à TRUE la propriété `Active` d'un composant `TTable`, vous obtenez le message "objet non trouvé', c'est que vous avez oublié d'exécuter l'Administrateur BDE afin de mettre à jour la configuration BDE.

Sécurité, les mots de passe

Vous avez entendu parler des pirates informatiques qui récupèrent des données top secret en décodant des informations repiquées sur une ligne de téléphone à l'aide d'un Cray 42. Il y a certes un peu d'exagération dans tout cela, mais la sécurité est un problème important qu'il ne faut pas négliger. La sécurité des données recouvre l'ensemble des opérations permettant à certains utilisateurs ou groupes d'utilisateurs d'accéder à des données en lecture ou en édition. Dans l'exemple des nombres premiers, que se passerait-il si n'importe qui pouvait accéder directement à la base de données (sans passer par l'application Delphi que vous avez amoureusement peaufinée) pour aller ajouter le nombre 42, ou 513 (513 non plus n'est pas premier) ?

Les niveaux de sécurité

Différents niveaux de sécurité sont disponibles selon le SGBD utilisé. Nous nous limiterons au cas des utilisateurs individuels, laissant de côté les groupes et les autres applications (les fonctionnalités sont identiques). Les niveaux de sécurité les plus courants sont :

- Aucun — Toute personne ayant physiquement accès aux données peut en faire ce qu'elle veut.
- Tout ou rien — Une fois que l'utilisateur a reçu le droit d'accéder à la base de données, il est libre d'y faire ce qu'il lui plaît.
- Multi-utilisateur, au niveau des tables — Chaque utilisateur peut accéder à des tables différentes dans différents modes. Ainsi, un administrateur pourra éditer ou modifier des enregistrement, tandis qu'un utilisateur sans privilèges ne pourra que consulter la table.
- Multi-utilisateur, au niveau des champs — Ce système est le plus souple : l'accès peut être défini, utilisateur par utilisateur, table par table, et champ par champ. Il est par exemple possible de ne voir que certains champs d'une table, et parmi ces champs avoir quelques droits de modification, mais pas d'effacement.

Authentification

Nouveau

Maintenant que vous savez définir des privilèges, utilisateur par utilisateur, le problème principal est de savoir que tel ou tel utilisateur tente d'accéder à la base. Pour vérifier que l'utilisateur est bien qui il prétend être, on a recours à l'authentification. Quand un utilisateur a prouvé son identité, on dit qu'il est authentifié (ou certifié).

Le moteur de bases de données doit également savoir authentifier un utilisateur et déterminer son type d'accès. La façon de procéder la plus courante repose sur des mots de passe. Paradox utilise un mot de passe maître qui définit ce propriétaire. Des mots de passe secondaires peuvent être définis par ce propriétaire, et il est possible de leur attribuer des accès par champ. Une autre méthode consiste à authentifier les utilisateurs par leur nom et un mot de passe. Cette méthode est meilleure car elle repose sur le nom de l'utilisateur et non sur un mot de passe. D'un point de vue administratif, le niveau de sécurité est plus satisfaisant puisque le propriétaire de la base n'a plus à connaître le mot de passe de chaque utilisateur.

Travailler avec plus d'une table

L'une des raisons majeures de la prééminence du modèle relationnel est qu'il est intuitif et facile à manipuler. Dans les exemples que nous avons étudiés jusqu'ici, nous n'utilisions qu'une table, qui contenait toutes les informations. Cependant, ce procédé ne peut être envisagé dans la majorité des situations. Considérons par exemple une université désirant garder une trace de ses étudiants et des cours qu'ils ont suivis. En se limitant à une seule table, on obtient un résultat similaire à celui du Tableau 11.11.

Tableau 11.10 : Représentation maladroite des informations relatives aux étudiants

Identifiant de l'étudiant	*Nom*	*Téléphone*	*Cours*	*Appréciation*
12345	Arthur Accroc	01-42-24-42-24	CH101	C
12345	Arthur Accroc	01-42-24-42-24	EN333	A
12345	Arthur Accroc	01-42-24-42-24	NP343	A
34464	Muad Dib	03-34-87-02-57	BI333	C
34464	Muad Dib	03-34-87-02-57	NP434	B

Cette représentation est inefficace car les données sont répétées d'un enregistrement à l'autre, et donc difficiles à gérer : si un étudiant change de numéro de téléphone, il faut retrouver tous les enregistrements qui lui sont relatifs et les modifier. D'autre part, la redondance systématique des informations est signe d'une mauvaise utilisation des mémoires de masse. S'il devenait nécessaire d'ajouter un descriptif de chaque cours, la table exploserait sans cause apparente.

Ce défaut vient de la conception de cette base de données qui ne prend pas en compte les relations existant entre les étudiants et les cours qu'ils suivent. Observez maintenant les deux tables suivantes qui contiennent autant d'informations que celle qui précède :

Tableau 11.11 : Table des étudiants

Identifiant de l'étudiant	*Nom*	*Téléphone*
12345	Arthur Accroc	01-42-24-42-24
34464	Muad Dib	03-34-87-02-57

Tableau 11.12 : Table des cours suivis par les étudiants

Identifiant de l'étudiant	*Cours*	*Appréciation*
12345	CH101	C
12345	EN333	A
12345	NP343	A
34464	BI333	C
34464	NP434	B

La conception de cette base est meilleure, les informations étant ventilées dans plusieurs tables.

Cette scission est rendue possible par les relations logiques existant entre les données des deux tables. Remarquez que, techniquement, rien n'empêche de travailler avec une seule table, mais ce serait ignorer la puissance du modèle relationnel.

Clés étrangères

Nouveau

Nous savons déjà qu'une clé est un champ ou une combinaison de champs permettant d'identifier de façon unique chaque enregistrement (les plus scientifiques d'entre nous diront qu'une clé est une injection de son domaine de valeurs vers celui de la table). On appelle ce type de clés des clés primaires. Certains champs seront qualifiés de clé étrangère. Une clé étrangère est un champ qui est la clé primaire d'une autre table. A titre d'exemple, l'identifiant de l'étudiant est la clé primaire de la table des étudiants, et une clé exportée dans la table des cours suivis par les étudiants. Notez bien que cet identifiant n'est pas une clé primaire dans la table des cours suivis par les étudiants.

Relations

Il y a trois types de relations entre deux tables. Les relations 1-1, n-1, et n-n. Examinons-les plus en détail. Les tables ayant entre elles une relation 1-1 sont rares, car il est souvent plus

pratique de les regrouper en une seule table. Cependant, si certaines informations sont rarement utilisées, il peut être pertinent de les stocker dans une table, et de ne mettre dans l'autre que celles dont on a fréquemment besoin. Une des relations les plus puissantes est la relation n-1. Pour bien comprendre ce qu'est une relation n-1, prenons un exemple :

Le service des cartes grises gère des informations sur les conducteurs et les automobiles. Les données relatives aux conducteurs sont stockées dans la table PILOTE, qui comprend un numéro de permis de conduire, un nom, une adresse, et un nombre de points. Les informations sur les véhicules sont stockées dans la table AUTO. Cette table comprend le numéro minéralogique de la voiture, son modèle, son année, et le numéro de permis de conduire de son propriétaire. Dans la table AUTO, le numéro de permis de conduire est une clé exportée, et le numéro minéralogique est la clé primaire. Dans la table PILOTE, le numéro du permis de conduire est la clé primaire. Un automobiliste peut posséder plus d'une voiture, mais une voiture ne peut avoir qu'un seul propriétaire. Il y a plusieurs (n) voitures pour un (1) automobiliste, c'est donc une relation n-1.

Il existe également des relations n-n. Les relations n-n sont souvent représentées à l'aide d'une table intermédiaire contenant les clés exportées à mettre en concordance. En voici une illustration :

Les étudiants suivent des cours. Chaque étudiant peut suivre plusieurs (n) cours. Symétriquement, un cours comprend plusieurs (n) étudiants. Il y a donc une relation n-n entre les étudiants et les cours. L'une des tables de cette base de données recensera les informations relatives aux étudiants (numéro de sécurité sociale, nom, coordonnées), une autre contiendra des détails sur les cours (identifiant du cours, intitulé, présentation, connaissances préalables). Pour effectuer le lien entre les deux, on ajoute une troisième table composée de deux clés exportées, à savoir le numéro de sécurité sociale de l'étudiant, et l'identifiant du cours. Cette dernière table peut servir indifféremment à générer la liste des étudiants d'un cours ou la liste des cours d'un étudiant.

L'objectif principal à garder en mémoire lors de la conception d'une base de données est la non-redondance des informations, ainsi que le respect de la cohérence intrinsèque des entités manipulées. Un autre point important à comprendre est qu'en matière de conception de bases de données, il n'y a pas de recette miracle. Alors que la programmation se compose souvent d'un enchaînement de figures imposées que le développeur a juste le loisir d'agencer, la conception de base de données se prête à autant de variantes qu'il existe d'individus. Nous allons maintenant nous pencher sur les méthodes permettant de manipuler facilement les tables ayant des relations n-1.

Intérêt des modules de données

Une des limitations de Delphi 1.0 résidait dans l'impossibilité d'enregistrer les relations entre différentes tables dans une unité séparée. Le développeur était donc obligé de redéfinir la relation entre les tables à chaque nouvelle application. Jusqu'à la fin de cette partie nous placerons les composants de base de données non visuels sur un module de données afin de pouvoir y accéder depuis plusieurs applications. N'oubliez pas que pour utiliser des composants d'un module de données vous devez choisir dans le menu Fichier l'option Utiliser unité.

Propriétés MasterFields et MasterSource

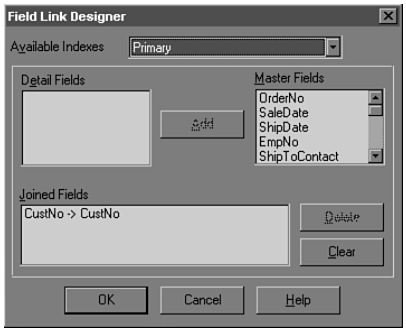
On a fréquemment besoin d'une relation n-1 pour fournir des informations détaillées sur un champ dans une table qui est clé exportée dans cette table. Delphi propose une méthode simple en utilisant les propriétés `MasterField` et `MasterSource` des objets `TTable`.

Les propriétés `MasterFields` et `MasterSource` permettent à une table de contrôler quel enregistrement est accessible dans une table détail. Par exemple, considérons une table contenant la liste des cours des étudiants. Il est possible, pour l'affichage, d'utiliser un composant DBGrid. Il pourrait également y avoir un ensemble de champs qui afficheraient des informations relatives à l'étudiant de l'enregistrement courant, comme son nom, son téléphone et ses coordonnées. Pour lier les tables ensemble, la table Studinfo utilise les propriétés `MasterSource` et `MasterField` afin de déterminer l'enregistrement courant de la table des cours. Dans cet exemple, la table des étudiants est la table détail, et celle des cours est la table maître.

Pour implémenter la relation maître/détail, vous devez associer une datasource à la table maître. Le nom de ce composant doit ensuite être spécifié dans le champ `MasterSource` de la table de détail. La dernière information nécessaire pour mettre la machine en branle est le nom du champ mis en commun dans les deux tables. Ce champ doit respecter certaines conditions. Le champ mis en commun doit être indexé dans la table détail. L'association de ce champ entre les deux tables se fait en utilisant la propriété `MasterField` de la table détail. Pour créer l'association, observez les étapes suivantes :

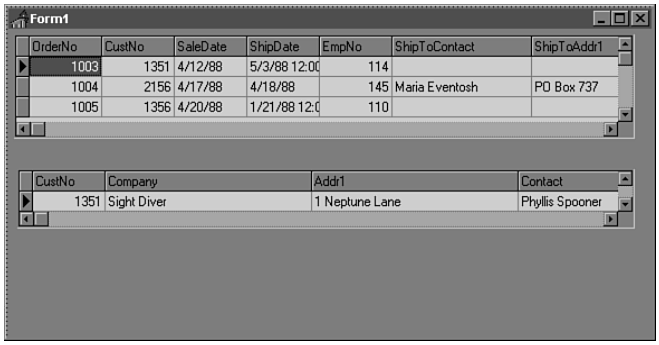
- Ajoutez des composants TTable et TDatasource pour les deux tables. Utilisez dans cet exemple ORDERS.DB comme table maître et CUSTOMER.DB comme table détail. Ces deux tables se trouvent dans la base de données DBDEMOS.
- Faites apparaître la feuille de propriétés de la table détail (CUSTOMER).
- Mettez dans la propriété `MasterSource` le nom du composant datasource qui accède à la table maître.
- Double-cliquez sur la propriété `MasterField` ; cela fait apparaître une boîte de dialogue. Il est nécessaire d'indiquer les champs à mettre en correspondance dans la table maître et dans la table détail. Les champs doivent être exactement du même type. Une fois ces champs liés, appuyez sur OK (voir Figure 11.13).

Figure 11.13
L'outil de gestion des liens.



L'application de la Figure 11.14 utilise un lien sur un champ afin d'afficher les informations relatives aux commandes dans un DBGrid et les informations relatives aux clients dans un autre. Chaque fois qu'une nouvelle commande est sélectionnée, le client associé apparaît.

Figure 11.14
Application utilisant des champs liés.



Delphi permet de manipuler des tables au moyen du composant `TTable`. Ce composant comporte des méthodes pour rechercher un enregistrement, trier les enregistrements, lier des tables, et se déplacer d'un enregistrement à l'autre. L'inconvénient du composant `TTable` est qu'il y a une méthode par action à effectuer, rendant complexe la combinaison des opérations. Pour régler ce problème, Delphi offre un autre moyen d'interagir avec des bases de données : le composant visuel `TQuery`. Ce composant permet à une application d'interagir avec une base de données au travers d'un langage dédié appelé SQL. SQL est un langage extrêmement puissant qui, bien utilisé, permet de simplifier prodigieusement des manipulations complexes.

Récapitulatif

Lors de cette journée, nous avons parcouru les généralités les plus importantes sur les bases de données, ainsi que leur mise en œuvre sous Delphi. Nous savons à présent utiliser un composant TTable pour avoir accès à une base de données. Le DataSource se comporte comme une interface entre l'objet TTable et les composants orientés données. Les composants orientés données sont un puissant jeu de composants permettant d'accéder à une table pour visualiser et manipuler les données. Nous avons également abordé les problèmes de performance, les clés et les index. Les clés sont utilisées pour s'assurer que chaque enregistrement est différenciable des autres, et l'intérêt principal des index est d'augmenter les performances. Nous avons également abordé les problèmes de sécurité et d'authentification, ainsi que les implémentations sous Paradox. Nous avons terminé en apprenant à utiliser plusieurs tables et des clés étrangères dans une application.

Atelier

L'atelier vous donne trois façons de vérifier que vous avez correctement assimilé le contenu de cette section. La section Questions - Réponses vous propose des questions couramment posées, ainsi que leurs réponses, la section Questionnaire vous pose des questions, dont vous trouverez les réponses en consultant l'Annexe A, et les Exercices vous permettent de mettre en pratique ce que vous venez d'apprendre. Tentez dans la mesure du possible de vous appliquer à chacune des trois sections avant de passer à la suite.

Questions - Réponses

Q Dans la table Studinfo, comment faire pour laisser l'utilisateur modifier tous les champs à l'exception du numéro de sécurité sociale ?

R Utilisez la propriété `ReadOnly` du composant StudInfoSSN. Les champs se manipulent comme des composants. Si la propriété `ReadOnly` vaut True, le champ ne peut être modifié.

Q Que se passe-t-il si une application tente d'outrepasser un contrôle de validité au niveau de la base ?

R Une exception se déclenche, il est du ressort du développeur de gérer ces situations avec des clauses `try` et `except`.

Q Comment gérer des relations n-1 ? Par exemple, stocker des informations relatives aux cours suivis par chaque étudiant ?

R La bonne méthode consiste à utiliser plusieurs tables. Un champ de la table détail sera la clé de la table maître. Ici, la table COURS comporterait un champ SSN qui assurerait le lien avec la table studinfo. Pour plus d'informations, voyez la journée suivante.

Q Je dispose d'une table locale que je dois sécuriser, mais je n'ai pas les outils pour le faire.

R Il faut gérer le problème à un autre niveau. Cryptez l'ensemble de la table, ou simplement les données sensibles. Une autre méthode consiste à stocker le fichier sur un réseau comportant ses propres permissions d'accès. N'oubliez pas que le niveau global de sécurité est déterminé par le maillon faible de la chaîne.

Questionnaire

1. Si une table est dans l'état dsBrowse, que se passe-t-il lorsque l'on tente d'exécuter une méthode d'édition telle que `post` ?
2. Comment déterminer le type des données stockées dans un champ ?
3. Quelle propriété faut-il modifier pour changer l'ordre de tri des enregistrements ?
4. Quels modèles de bases de données sont accessibles par plusieurs personnes ou plusieurs applications en même temps ? Lesquels ne le sont pas ?
5. Si une table est dans l'état `dsCalcFields`, qu'est-il impossible de faire avec les champs non calculés ?
6. Dans une relation n-1, quel est le type de la clé de la colonne de la table "n" qui indique la clé de la table "1" ?

Exercices

1. Créez une application qui utilise tous les contrôles orientés données.
2. Modifiez l'exemple des nombres premiers et utilisez un champ calculé pour afficher le carré de chaque nombre.
3. Ajoutez un contrôle de validité pour empêcher de rentrer des nombres trop grands dans la table des nombres premiers. Ajoutez du code à l'application de façon à afficher un message d'erreur quand une valeur trop grande est saisie.

12

Etats

Après avoir appris à développer des applications de bases de données sous Delphi, ainsi que d'autres applications capables de traiter des données, il est temps de s'intéresser aux états, schémas et graphiques que l'on peut tirer de ces données pour impression ou visualisation. Cette journée s'intéresse plus spécifiquement aux composants QuickReport et DecisionCube. Notre but n'est pas de vous entraîner dans les arcanes de ces composants, mais plutôt de vous en donner un aperçu à partir duquel vous pourrez plus tard approfondir les voies qui vous semblent les plus prometteuses. Ces outils ont été développés par des sociétés tierces et inclus dans Delphi avec toutes leurs fonctionnalités. Il n'est pas obligatoire de les enregistrer si vous voulez distribuer vos applications, mais vous pouvez le faire si vous désirez acheter les versions suivantes de ces produits.

QuickReport

Qu'est-ce au juste que QuickReport ? QuickReport pour Delphi est un ensemble de composants qui vous permettent de concevoir vos états visuellement et de les rattacher à votre code. Le produit fini permet à vos utilisateurs de générer des états de bonne facture comportant du texte et des graphiques. QuickReport est un générateur d'états par bandes qui dispose de nombreuses fonctionnalités.

En voici quelques-unes :

- Conception visuelle d'états
- Composants VCL entièrement écrits en Pascal Objet
- Compatibilité avec Delphi 1, 2 et 3
- Multithread
- Prévisualisation immédiate
- Etats de longueur illimitée
- Champs mémo de longueur illimitée
- Utilisation de n'importe quelle source de données, y compris ODBC
- Conception des états par l'utilisateur final
- Prévisualisation personnalisable
- Utilisation d'autres composants imprimables tels que des schémas, des codes barre, des formats graphiques étendus, et plus encore
- Evaluateur d'expressions sophistiqué et concepteur d'expressions
- Expert Etat
- Exportation des états aux formats ASCII, délimités par des virgules et HTML, ou filtres d'exportation personnalisés

Ces informations sont extraites de la documentation QuickReport fournie sur le CD-ROM Delphi. Ce ne sont que quelques points forts du produit. La mission de cet ouvrage n'est pas d'examiner en détail chacune des fonctionnalités de cet outil. Nous vous conseillons donc d'étudier

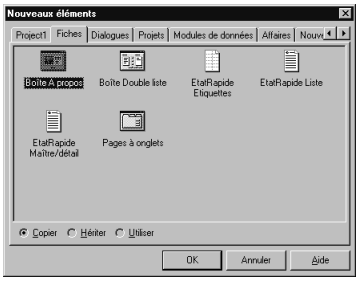
la documentation fournie après avoir terminé la lecture de ce partie, qui vous donnera cependant un premier contact ainsi que les informations nécessaires pour créer un état simple.

Modèles QuickReport

Delphi comprend plusieurs modèles et un expert afin de vous permettre de créer rapidement des états simples. Plutôt que de nous attarder sur ces modèles, nous allons travailler directement sur un exemple afin de comprendre le fonctionnement sous-jacent de l'application. Une fois que vous aurez acquis cette première expérience de QuickReport, vous serez à même d'utiliser les modèles et l'expert sans aucune difficulté.

Sélectionnez Fichier|Nouveau, et, dans la boîte de dialogue Nouveaux éléments, cliquez sur l'onglet Fiches. Vous verrez apparaître les icônes QuickReport Etiquettes, QuickReport Liste et QuickReport Maître/détail (comme indiqué Figure 12.1).

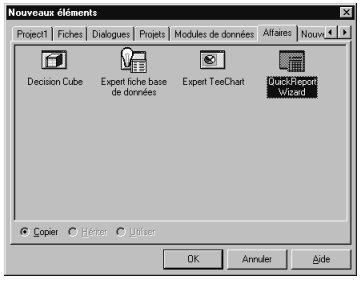
Figure 12.1
Modèles QuickReport.



Ce sont ces modèles que vous utiliserez pour créer rapidement des états. Il suffit de les ajouter à votre projet et de modifier leurs paramètres en fonction de vos besoins.

L'onglet Affaires de la boîte de dialogue Nouveaux éléments comporte des Experts fiche du monde de l'entreprise, comme indiqué Figure 12.2. Après avoir lu cette partie, vous aurez assez de connaissances pour utiliser ces modèles ainsi que l'expert. Pour l'instant, occupons-nous de quelques-uns des composants QuickReport.

Figure 12.2
L'onglet Affaires et l'icône Expert QuickReport.



Les composants QuickReport

Vous avez eu au Jour 8 une brève description des composants VCL, y compris les composants QuickReport. Ceux-ci mettent à disposition de quoi créer rapidement des états dans vos applications Delphi. Commençons par regarder un peu plus en détail certains d'entre eux et mettons-les en œuvre en écrivant une petite application destinée à imprimer un état à partir d'une des bases de données d'exemple fournies avec Delphi.

Nouveau

QuickReport étant un générateur d'états par bandes, il est important que vous compreniez ce que sont ces bandes et comment les utiliser pour construire vos états. Les bandes sont des zones d'un état qui vous permettent de concevoir visuellement celui-ci en créant des sections contenant du texte, des images, des graphiques, des schémas, etc. Les bandes sont des conteneurs pour d'autres composants qui apportent l'information ou les graphiques dans l'état. Vous pouvez activer ou désactiver certaines de ces bandes en réglant des propriétés du composant `TQuickRep`*, et en ajouter d'autres en les faisant glisser depuis la palette des composants dans le composant* `TQuickRep`*. Dans tous les cas, les bandes représentent un élément capital de la conception de votre état.*

Le composant *TQuickRep*

Le composant `TQuickRep` est utilisé pour se connecter à un ensemble de données grâce à sa propriété `Table`. Il comporte également d'autres propriétés permettant de contrôler finement la création de vos états, y compris le titre, les bandes, les polices, le paramétrage de l'impression, etc. Ce composant est également un composant visuel ; une fois connecté à la base de données, il est utilisé comme conteneur pour les bandes qui constituent l'état.

Les composants de bande

Les composants `QRSubDetail`, `QRBand`, `QRChildBand` et `QRGroup` sont les composants que vous pouvez ajouter aux états afin d'insérer des titres, des en-têtes, des pied de page, des zones de détail pour les données de l'état, etc. En connectant ces bandes à une base de données et en les disposant sur l'état là où vous voulez afficher l'information, vous pouvez rapidement concevoir vos états (reportez-vous à l'aide en ligne pour plus d'informations).

Autres composants QuickReport

Les autres composants se répartissent en deux catégories : ceux qui sont orientés données, et ceux qui ne le sont pas. Ces composants sont utilisés pour placer du texte, des images, des graphiques, etc., dans les bandes de l'état. Vous en apprendrez plus à leur sujet en créant un état.

Créer un état simple

Nous allons construire un état d'exemple qui affichera du texte et une image, afin que vous puissiez apprécier la puissance de QuickReport. Lancez Delphi, créez un nouveau projet, et effectuez les étapes suivantes :

1. Modifiez la propriété `Caption` de la fiche pour lui donner le nom `Générateur d'état de base de données`, et définissez sa propriété `Name` à `RepGen`.
2. Ajoutez une seconde fiche et affectez à sa propriété `Caption` la valeur `Fiche d'état - invisible`. Sa propriété `Name` doit valoir `RepForm`. Assurez-vous que `Visible` est à `False`, car l'utilisateur final ne doit pas voir cette fiche : elle servira de conteneur aux composants QuickReport.
3. Prenez un composant `TTable` et mettez-le sur la fiche `RepForm`.
4. Utilisez l'une des bases de données de démonstration fournies avec Delphi en définissant la propriété `DatabaseName` de `TTable` à `DBDemos`.
5. Définissez la propriété `TableName` de `TTable` à `ANIMALS.DBF`.
6. Définissez la propriété `Active` de `TTable` à `True`.
7. Dans l'onglet `QReport`, sélectionnez un composant `QuickRep` et mettez-le sur la fiche `RepForm`.
8. Définissez la propriété `DataSet` du composant `QuickRep` à `Table1`.
9. Double-cliquez sur la propriété `Bands` du composant `QuickRep` afin d'afficher ses sous-propriétés, et affectez `True` à `HasDetail`. Vous verrez alors la bande détail de l'état.
10. Mettez un composant `QRLabel` dans la bande détail du composant `QuickRep`, et affectez `Nom d'animal` à sa propriété `Caption`.
11. Mettez un composant `QRDBText` à droite du composant `QRLabel`, et affectez `Table1` à sa propriété `Dataset`, ainsi que `Name` à sa propriété `DataField`.
12. Cliquez sur la bande détail et augmentez sa hauteur afin de pouvoir y faire tenir une image.
13. Ajoutez un composant `QRDBImage` sur la bande détail, et placez-le sur la gauche afin de vous assurer qu'il forme un carré qui tient dans la bande. Redimensionnez la bande et ce composant si nécessaire.
14. Affectez à la propriété `Dataset` du composant `QRDBImage` la valeur `Table1`, et mettez `BMP` dans sa propriété `DataField`.

Si tout s'est bien passé, vous devriez avoir un premier aperçu de votre état en cliquant avec le bouton de droite sur le composant `QuickRep` et en sélectionnant dans le menu contextuel l'option Prévisualiser. Une fenêtre apparaît alors, qui vous donne une prévisualisation de votre état, comme indiqué Figure 12.3.

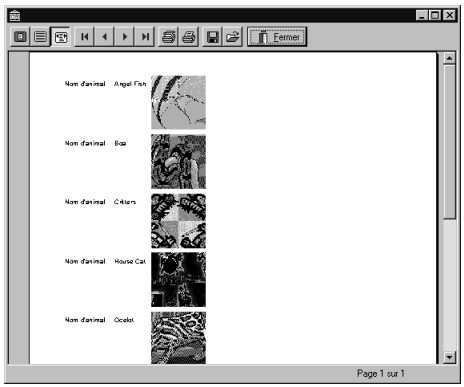


Figure 12.3
La fenêtre de prévisualisation de QuickReport.

Félicitations ! Vous venez de créer votre premier état sans une seule ligne de code. Si vous le désirez, il est possible d'imprimer un état à partir de la fenêtre de prévisualisation (pour ce faire, cliquez sur l'icône d'impression de la barre d'outils).

Vous pouvez utiliser ce que nous avons vu ensemble pour ajouter d'autres champs à votre état en ajoutant d'autres composants. Voyons maintenant comment ajouter des fonctionnalités disponibles lors de l'exécution.

Ajouter une fenêtre de prévisualisation

Ajouter une fenêtre de prévisualisation est un jeu d'enfant. Il vous suffit d'ajouter un composant bouton sur l'une des fiches accessibles à l'utilisateur. Dans notre exemple, il s'agit de la fiche `RepGen`. Ensuite, double-cliquez sur le bouton que vous venez d'ajouter. La fenêtre d'édition de code apparaît alors. Ajoutez le code suivant dans l'événement `OnClick` :

```
RepForm.QuickRep1.Preview
```

Si vous tentez d'exécuter, de compiler ou de vérifier la syntaxe du programme, Delphi détecte qu'il manque une unité dans la clause `Uses` de la première fiche et vous demande si vous désirez l'ajouter. Répondez Oui, et la clause `Uses` sera automatiquement corrigée.

C'est tout ! Si vous exécutez l'application et cliquez sur le bouton de prévisualisation, l'écran de prévisualisation apparaît. Vous voyez bien que c'est facile. Les utilisateurs seront persuadés que vous avez passé des heures à mettre en place cette fonctionnalité, alors qu'il ne vous aura fallu que quelques minutes.

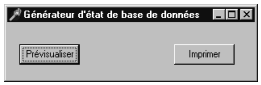
Imprimer des états

Au cas où vous ne l'auriez pas déjà deviné, il est aussi facile d'imprimer un état que d'ajouter un bouton de prévisualisation. Ajoutez un autre bouton à la fiche, et affectez `Imprimer` à sa propriété `Caption`. Double-cliquez sur le bouton, et ajoutez le code suivant au Gestionnaire d'événements `OnClick` :

```
RepForm.QuickRep1.Print
```

La fiche principale de votre application devrait à présent ressembler à celle de la Figure 12.4, et le code à celui du Listing 12.1. Naturellement, la fiche cachée comporte également du code, mais nous ne le donnons pas ici puisqu'il a été entièrement généré par Delphi.

Figure 12.4
L'application Etat.



Listing 12.1 : La fiche principale de l'application Etat

```
unit main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs,
  StdCtrls;

type
  TRepGen = class(TForm)
    Button1: TButton;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    {  déclarations privées }
  public
    {  déclarations publiques }
  end;

var
  RepGen: TRepGen;

implementation

uses qrep;
```

```
{$R *.DFM}

procedure TRepGen.Button1Click(Sender: TObject);
begin
     RepForm.QuickRep1.Preview;
end;

procedure TRepGen.Button2Click(Sender: TObject);
begin
     RepForm.QuickRep1.Print;
end;
end
```

Analyse *Si vous ne l'avez pas déjà fait, exécutez l'application et testez-la. Cliquez sur le bouton Prévisualiser afin de faire apparaître la fenêtre de prévisualisation, ou sur le bouton Imprimer pour imprimer l'état. La base de données utilisée dans cet exemple ne comportant que quelques enregistrements, l'impression ne comportera que deux ou trois pages. La fenêtre de prévisualisation a l'avantage de tenir compte des données réelles et donc d'afficher le nombre de pages à imprimer, que vous pouvez parcourir avant impression, etc. Vos utilisateurs pourront donc avoir accès à un grand nombre de fonctionnalités sans qu'il vous en ait coûté plus de deux lignes de code.*

Decision Cube

Si vous disposez de l'édition CS de Delphi, les composants Decision Cube vous permettront de faire de l'analyse de données multidimensionnelle dans vos applications.

Les composants Decision Cube

Ces composants sont au nombre de six. Trois d'entre eux permettent de communiquer avec le moteur de base de données de Delphi (`DecisionCube`, `DecisionQuery` et `DecisionSource`), les trois autres étant des composants visuels vous permettant de travailler avec les données et de les afficher (`DecisionPivot`, `DecisionGrid` et `DecisionGraph`).

Aperçu de Decision Cube

Grâce aux composants Decision Cube, vous pouvez créer des applications permettant aux utilisateurs de créer des états complexes. Ils peuvent être intégrés dans une application, ou distribués sur un réseau (par exemple au travers d'une page Web). Ces composants n'étant fournis qu'avec la version CS de Delphi, nous n'en parlerons pas plus longuement ici. Reportez-vous à l'aide en ligne pour en savoir plus.

Récapitulatif

Cette journée d'apprentissage vous a permis de découvrir QuickReport et la puissance qu'il apporte à Delphi. Cet outil vous fera gagner des heures de développement lors de la création d'états d'apparence professionnelle. Nous avons étudié les principales fonctionnalités de Quick Report et créé une application d'exemple à partir d'une des bases de données fournies avec Delphi (`ANIMALS.DBF`). Vous avez également bâti une petite application en utilisant deux lignes de code afin de prévisualiser et d'imprimer les états.

Nous n'avons bien évidemment pas abordé en profondeur ces différents composants. Vous devriez quand même avoir une idée précise de leurs fonctionnalités, ce qui vous permettra de les utiliser, en approfondissant vos connaissances grâce à l'aide en ligne.

Nous avons terminé par un rapide survol de Decision Cube, un ensemble d'outils vous permettant de créer des outils et des états d'analyse de données multidimensionnelle. Decision Cube n'est fourni qu'avec la version CS de Delphi 3.

Atelier

L'atelier vous donne trois moyens de vérifier que vous avez correctement assimilé le contenu de cette partie. La section Questions - Réponses reprend des questions posées couramment, et y répond, la section Questionnaire vous pose des questions, dont vous trouverez les réponses en consultant l'Annexe A, et les Exercices vous permettent de mettre en pratique ce que vous venez d'apprendre. Tentez dans la mesure du possible de vous appliquer à chacune des trois sections avant de passer à la suite.

Questions - Réponses

Q Comment ajouter des graphiques à mes états QuickReport ?

R Utilisez le composant `TeeChart QRChart`.

Q QuickReport peut-il faire des étiquettes ?

R L'un des modèles QuickReport sert justement à générer des étiquettes. Grâce à sa conception par bandes des états, QuickReport est parfaitement adapté à un grand nombre de situations.

Questionnaire

1. Quels sont les types d'états générés par QuickReport ?
2. A quoi les bandes servent-elles dans QuickReport ?
3. L'utilisateur voit-il le composant `QuickReport` ?

4. Lors de la construction d'une application utilisant QuickReport, comment se connecte-t-on à la base de données ? En d'autres termes, quels composants faut-il utiliser ?

Exercice

1. Créez une application affichant un état pour chaque champ de la base de données Animals que nous avons utilisée dans cette section (référez-vous à la documentation de QuickReport si vous avez besoin de plus de renseignements).

13

Créer des composants visuels et ActiveX

A mesure que le logiciel devient toujours plus sophistiqué et complexe, l'utilisation de composants s'impose pour le développement d'applications. Delphi est un outil de développement parfait pour créer des composants servant à n'importe quelle application logicielle qui en utilisent. Vous pouvez ainsi développer des composants visuels Delphi natifs et ActiveX. Les premiers peuvent être construits et installés dans l'EDI Delphi, sans les complications associées aux ActiveX. Les seconds peuvent être importés dans Delphi, Visual Basic, dans des pages Web, et dans tout autre produit capable de les incorporer.

Nous commencerons cette journée par un aperçu des avantages liés à l'utilisation de composants, puis nous verrons comment créer des composants visuels en Delphi, en commençant par un exemple très simple pour finir sur des composants utiles et puissants. Nous verrons ensuite la manière dont Delphi peut convertir des composants visuels en ActiveX.

Pourquoi écrire des composants ?

La puissance des composants tient au fait qu'ils cachent au développeur tous les détails de l'implémentation. De même qu'un conducteur n'a pas besoin, pour conduire, de connaître les principes thermodynamiques à l'œuvre dans un moteur à explosion, un développeur d'application n'a pas besoin de savoir comment fonctionne un composant pour l'utiliser. Il lui suffit de savoir comment communiquer avec lui. Il y a au moins quatre bonnes raisons d'écrire vos propres composants.

Réutiliser le code

L'interface vers un composant est intégrée à l'environnement de développement Delphi. Par conséquent, si vous utilisez fréquemment un objet, vous pouvez le transformer en composant visuel, l'ajouter à la barre d'outils, ce qui en facilite l'utilisation et en cache l'implémentation au sein de la bibliothèque.

Modifier les composants visuels existants

Comme Delphi est un langage orienté objet et comme un composant visuel est un objet, vous pouvez créer un composant visuel comme sous-classe d'un composant déjà existant. Supposons, par exemple, que vous utilisiez souvent des cercles bleus dans vos applications. Vous voulez tirer parti de toutes les propriétés, événements et méthodes du composant `TShape`, mais vous prenez toujours un cercle comme paramètre pour la forme et la couleur bleue. Vous pouvez dès lors créer une sous-classe de `TShape` appelée `TCercleBleu`. La classe `TCercleBleu` aura, par défaut, la propriété `shape` égale à `circle` et la propriété `color` égale à `Blue`.

Vendre des composants

Si vous devez ajouter des fonctionnalités spécialisées à votre application, vous pouvez acheter un composant à un éditeur indépendant. C'est l'un des plus grands avantages d'un système

basé sur les composants. Ainsi, si vous avez besoin de fonctions de réseau, vous pouvez écrire vous-même les routines afférentes ou acheter un ensemble de composants qui se chargent de ces fonctionnalités. De même, vous pouvez créer un composant et le vendre à d'autres développeurs. Lorsque vous vendez la bibliothèque, vous ne fournissez qu'une version compilée du produit. Votre client ne voit donc pas son code source ou les détails de son implémentation. Cependant, lorsque le composant est ajouté à une fiche, votre acheteur peut communiquer avec le composant par le biais de l'IDE.

Les modifications de comportement en cours de développement

Si vous compilez du code pour en faire un composant visuel, vous pouvez voir son aspect se modifier au cours du développement. Ainsi, si vous définissez comme un cercle la propriété `Shape` du composant `TShape`, la forme sur la fiche se transforme en cercle, et ce, avant que le composant ne soit compilé. Cela est très utile dans le cas de composants qui sont visibles dans l'application. Vous pouvez ainsi savoir à quoi celle-ci ressemblera une fois compilée.

L'ancêtre des composants : quelques mots sur les DLL

Avant d'étudier ensemble la création de composants, nous allons voir la construction de DLL en Delphi. Les DLL ne sont pas orientées objet. Elles vous permettent simplement de développer une bibliothèque de procédure dans un langage, utilisable dans n'importe quel autre. Un autre grand avantage des DLL est qu'elles sont chargées en phase d'exécution au lieu d'être liées statiquement dans l'application. Ceci vous permet de diviser le développement et la distribution en plusieurs composants, au lieu de développer et de lier statiquement une seule grosse application.

Nous allons créer une DLL simple que nous appellerons à partir d'un programme C. Elle contiendra une procédure acceptant un paramètre (un `PChar`), qui inverse l'ordre de la chaîne ainsi passée. De cette façon, si vous appelez `ReverseStr(X)` avec `X = "ZORGLUB"`, `X` sera égal à `"BULGROZ"` après appel à la procédure.

Pour créer une nouvelle DLL, suivez ces étapes :

1. Choisissez Fichier|Nouveau dans le menu principal, puis DLL. Un squelette de DLL est alors créé.
2. Ajoutez les fonctions et procédures à implémenter. Dans notre exemple, on ajoute le code d'implémentation pour la procédure `ReverseStr`. (voir Listing 13.1).

Listing 13.1 : ReverseStr, une DLL qui inverse les chaînes

```
library DelphiReverse;

{ Remarque importante à propos de la gestion mémoire des DLL : ShareMem doit
  être la première unité dans la clause USES de votre bibliothèque,
  ET dans la clause USES de votre projet (sélectionnez Voir-Source du projet)
  si vos DLLs exportent des procédures ou des fonctions passant en paramètre
  des chaînes ou des résultats de fonction.
  Ceci s'applique à toutes les chaînes passées par ou à vos DLL - mêmes
  celles
  qui sont imbriquées dans des enregistrements ou des classes. ShareMem est
  l'unité d'interface du gestionnaire de mémoire partagée DELPHIMM.DLL, qui
  doit être déployée avec vos propres DLL. Pour éviter l'emploi de DELPHIMM
  DLL,
  passez vos chaînes en utilisant des paramètres PChar ou ShortString. }
uses
  SysUtils,
  Classes;

procedure ReverseStr(strToReverse : PChar);export;stdcall;

var
   BFLen : integer;
   Temp  : char;
   Count : integer;

begin
   BFLen := StrLen(strToReverse);
   for Count := 0 to (BFLen div 2)-1 do
   begin
      Temp := strToReverse[Count];
      strToReverse[Count]   := strToReverse[BFLen - Count - 1];
      strToReverse[BFLen - Count - 1] := Temp;
   end;
end;

exports
  ReverseStr;
end.
```

Analyse

Ce programme implémente les fonctions nécessaires. Vous pouvez remarquer qu'il n'est pas nécessaire de placer les options `export` *et* `stdcall` *à la suite de la déclaration de procédure. Celles-ci spécifient que la fonction doit être rendue disponible et qu'une convention d'appel standard doit être utilisée pour passer des paramètres. La seule autre chose qui doit figurer dans le source est la section* `exports` *qui se trouve au bas de l'unité, car elle spécifie que la procédure* `ReverseStr` *doit être exportée. En exportant une fonction on la rend visible au monde extérieur. Si vous ne le faites pas, elle se comporte comme une fonction interne. De même que la section* `interface` *d'une unité permet d'utiliser des fonctions en dehors de celle-ci, la clause* `exports` *permet à une fonction d'être appelée en dehors de la DLL.*

Une fois le projet compilé, vous disposez d'une DLL qui peut être appelée à partir de pratiquement n'importe quel langage, que ce soit C, C++, Visual Basic et bien sûr Delphi. Le Listing 13.2 montre comment un programme C peut charger dynamiquement la DLL `DelphiReverse` pour appeler la procédure `ReverseStr`.

Listing 13.2 : Le code C appelant la DLL Delphi

```
#include <stdio.h>
#include <windows.h>

void main()
{
HANDLE  TheLib;
FARPROC ReverseStr;
char    TheStr[] = "Elu par cette crapulE";
  TheLib = LoadLibrary("DelphiReverse");
  ReverseStr =  GetProcAddress(TheLib,"ReverseStr");
  printf("Chaine initiale:%s\n",TheStr);
  ReverseStr(TheStr);
  printf("Chaîne inversée :%s\n",TheStr);
  FreeLibrary(TheLib);
};
```

Analyse

Ce programme déclare une chaîne et lui affecte "Elu par cette crapulE". Il charge ensuite la DLL `DelphiReverse` *en mémoire et obtient l'adresse de la procédure* `ReverseStr`*. Le programme affiche la chaîne initiale, l'inverse, puis l'affiche une fois inversée.*

La sortie de ce programme est la suivante :

```
D:\BBTD30\Samples\CH14\CallDll\Debug>calldll
Chaîne initiale : Elu par cette crapulE
Chaîne inversée : Eluparc ettec rap ulE
```

Construire et installer un composant

Il est facile de créer des DLL, mais elles ne s'intègrent pas à l'EDI de Delphi. Elles sont utiles, mais ne sont pas orientées objets. Les composants visuels ne comportent pas ces deux défauts. Le premier que vous allez construire dans cette partie se compile simplement et s'installe ou s'enlève de la barre d'outils. Vous pouvez aussi l'ajouter à une fiche, mais il ne fera strictement rien.

Dans Delphi, les composants sont compilés sous forme de paquets. En conséquence, la première chose à faire est de créer un nouveau paquet. Choisissez Fichier, Nouveau pour faire apparaître la boîte de dialogue Nouveaux éléments, puis sélectionnez l'icône "paquet". Un nom de fichier vous sera alors demandé. Dans les exemples qui suivent, nous utiliserons le nom `tdcomps`. Vous devriez maintenant voir une boîte de dialogue Gestionnaire de paquet vierge (voir Figure 13.1).

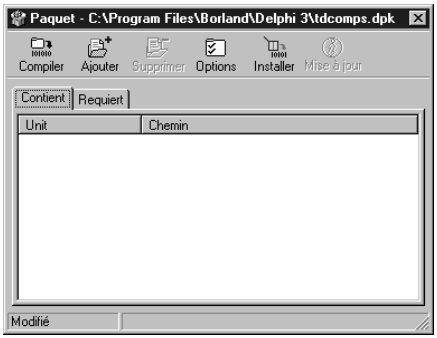


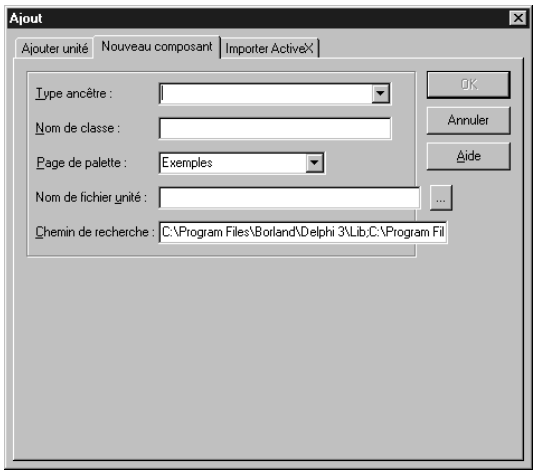
Figure 13.1
Le Gestionnaire de paquet prêt à ajouter un nouveau composant.

Ajouter le composant *TDoNothing* au paquet

Dans cet exemple, nous allons utiliser l'Expert composant pour générer le code nécessaire à la création du squelette d'un composant. Tous les composants doivent être issus d'autres composants. Si vous souhaitez partir de zéro, vous devez créer une sous-classe de `TComponent`, comme suit :

1. Cliquez sur l'icône Ajouter dans la boîte de dialogue Gestionnaire de paquet. La boîte Ajouter apparaît alors.
2. Sélectionnez l'onglet Nouveau composant.
3. Spécifiez `TComponent` comme type d'Ancêtre, `TDoNothing` comme Nom de classe et `Exemples` comme Page de palette. Pour le nom du fichier d'unité, choisissez un nouveau fichier .pas qui sera utilisé pour le code source, et laissez la valeur par défaut dans le Chemin de recherche. Votre boîte de dialogue devrait alors ressembler à celle de la Figure 13.2.
4. Cliquez sur OK pour enregistrer ces paramètres.

Figure 13.2
L'Expert composant pour TDoNothing.



Delphi crée le noyau d'une unité qui se compile en un composant visuel. En double-cliquant sur celle-ci dans le Gestionnaire de paquet, vous faites apparaître le code source. Le Listing 13.3 montre le code que l'Expert génère pour le composant `TDoNothing`.

Listing 13.3 : Le squelette d'un composant visuel

```
unit DoNothing;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs;

type
  TDoNothing = class(TComponent)
  private
    { Déclarations privées }
  protected
    { Déclarations protégées }
  public
    { Déclarations publiques }
  published
    { Déclarations publiées }
```

```
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Samples', [TDoNothing]);
end;

end.
```

Analyse *En temps normal, vous modifiez le noyau pour ajouter vos propres fonctionnalités au composant. La bibliothèque de composants est constituée de la classe* `TDoNothing` *et de la procédure* `Register`*. Delphi utilise tout le code associé à la classe pour déterminer le fonctionnement du composant. Il appelle également la procédure* `Register` *pour placer le composant sur la barre d'outils. En l'occurrence, Delphi se contente de l'installer sur l'onglet Exemples.*

Compiler et installer le paquet et les composants

Pour compiler et installer le composant, suivez ces étapes :

1. Dans le Gestionnaire de paquet, cliquez sur l'icône Compiler. L'unité sera alors compilée dans le fichier de paquet DPK.
2. Dans le Gestionnaire de paquet, cliquez sur l'icône Installer. Le paquet sera alors installé. Si son installation fonctionne bien, vous en serez averti par une boîte de dialogue (voir Figure 13.3).

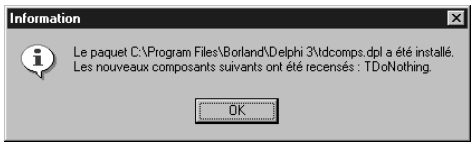
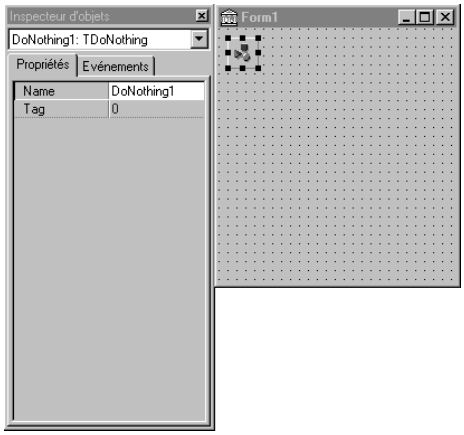


Figure 13.3
Le paquet a été installé avec succès.

Regardez la barre d'outils Exemples. Vous pouvez voir qu'un nouveau composant y est apparu. Placez le pointeur de la souris dessus pour qu'une info-bulle apparaisse. Le nouveau composant est `DoNothing`. Créez une nouvelle application et ajoutez-le à la fiche. Lorsque vous cliquez dessus pour visualiser les pages de propriétés et d'événements, vous voyez qu'il n'y a que deux propriétés (voir Figure 13.4). La propriété `Name` est `DoNothing1` par défaut et `Tag` est `0` par défaut. La page d'événements n'en contient aucun. C'est un composant visuel, dont la fonction est de ne rien faire.

Figure 13.4
L'application et la page de propriétés avec le composant DoNothing.



Enlever le composant

Il arrive que vous ayez besoin de retirer un composant de la barre d'outils ou de le désinstaller de Delphi. Cela peut être le cas si vous recevez une version mise à jour d'un composant visuel, ou si vous souhaitez simplement enlever un composant dont vous n'avez pas l'utilité. `TDoNothing` est par essence inutile. Pour l'enlever, suivez ces étapes :

1. Sélectionnez Projet, Options dans le menu, puis sélectionnez l'onglet Paquets.
2. Dans cet onglet, cliquez sur le paquet que vous venez d'installer, puis cliquez sur Supprimer. Le composant `TDoNothing` est alors retiré de la barre d'outils.

Ecrire un composant visuel

Un développeur d'applications utilise des composants en les plaçant sur une fiche ou en employant la méthode `Create`. Une fois le composant créé, vous pouvez le manipuler en définissant des propriétés, en appelant des méthodes et en répondant à des événements. Vous n'avez pas à vous soucier de la façon dont fonctionnent les propriétés et les méthodes. Le composant appelle son Gestionnaire d'événements lorsqu'un événement particulier survient. Lorsque vous écrivez un composant, vous devez détailler l'implémentation pour les propriétés et les méthodes, et appeler des Gestionnaires d'événements. Pour ce faire, définissez une classe qui devient le composant et complétez les différents éléments.

Déclarations privées, protégées, publiques et publiées

Delphi utilise des classes d'objet pour créer des composants visuels. Les différentes parties de la définition de la classe d'objet sont déclarées dans plusieurs régions protégées. Les variables, les procédures et les fonctions peuvent avoir quatre types d'accès :

`Privé (Private)`	Seules les procédures et les fonctions définies dans la définition de classe ont un accès, et seules les routines se trouvant dans la même unité en ont également un.
`Protégé (Protected)`	Les procédures et fonctions définies dans la définition de classe, ainsi que les procédures et fonctions des classes descendantes, en ont un.
`Public (Public)`	Toutes les procédures et fonctions ont un accès.
`Publié (Published)`	Accès public avec un branchement sur l'EDI de Delphi permettant d'afficher les informations dans les pages propriétés et événements.

Propriétés

Un développeur Delphi utilise les propriétés d'un composant pour lire ou modifier certains attributs, qui sont similaires aux champs de données stockés dans une classe. Les propriétés peuvent cependant provoquer l'exécution de code. Ainsi, lorsque vous modifiez la propriété `Shape` du composant `TShape`, ce dernier change de forme. Il existe un mécanisme qui lui ordonne de se modifier lorsque la propriété change. Autrement dit, une propriété peut endosser deux rôles. Ce peut être une donnée affectant le fonctionnement d'un composant, ou le déclencheur d'une action.

Méthodes

Les méthodes sont des procédures et des fonctions qu'une classe a rendues publiques. Bien que vous puissiez utiliser des propriétés pour appeler une fonction ou une procédure, ne le faites que si c'est logique. En revanche, les méthodes peuvent être utilisées n'importe quand. Celles-ci peuvent accepter plusieurs paramètres et renvoyer des données par le biais de déclarations de variables VAR, alors que les propriétés sont définies avec une donnée.

Evénements

Les événements permettent au développeur ou à l'utilisateur d'améliorer le composant lorsqu'un événement survient. Ainsi, l'événement `OnClick` signifie "Si vous voulez faire quelque chose lorsque l'utilisateur clique ici, dites-moi quelle procédure exécuter pour cela". C'est le travail du concepteur d'appeler les événements du composant si nécessaire.

Construire un composant utile : *TMult*

Nous allons maintenant créer un composant visuel qui effectue une tâche et comprend au moins une propriété, une méthode et un événement. Il est assez sommaire, et son intérêt est avant tout didactique. Cela montrera aussi comment le dériver à partir du sommet de la hiérarchie des composants. Tous les composants ont `TComponent` dans leur arbre généalogique. `TMult` est un descendant direct de `TComponent`.

Créer *TMult*

Le composant `TMult` a deux propriétés de type `integer` qui peuvent être définies au moment de la conception ou de l'exécution. Il a une méthode, `DoMult`. Lorsque celle-ci est exécutée, les deux valeurs des propriétés sont multipliées et le résultat est placé dans une troisième propriété appelée `Res`. Un événement, `OnTooBig` est aussi implémenté. Si l'un des deux nombres est défini comme étant supérieur à 100 lorsque la méthode `DoMult` est appelée, le composant appelle le code vers lequel l'utilisateur à pointé dans la page d'événements. `TMult` est un exemple de composant purement fonctionnel : il n'a pas de composante graphique. Parmi les composants standard du même type, on peut citer `TTimer`, `TDataBase` et ceux de `Table`.

`TMult` contient les propriétés suivantes :

`Val1`	La première valeur à multiplier. Elle est disponible lors de la conception et de l'exécution.
`Val2`	La deuxième valeur à multiplier. Elle est disponible à la conception et à l'exécution.
`Res`	La valeur obtenue en multipliant `Val1` et `Val2`. Disponible seulement en cours d'exécution.

`TMult` contient une méthode, `DoMult`, qui implémente la multiplication de `Val1` par `Val2`. L'événement de `TMult`, `OnTooBig`, appelle le Gestionnaire d'événements de l'utilisateur (s'il existe) lorsque `Val1` est multipliée par `Val2` et qu'une des deux valeurs est supérieure à 100.

Construire *TMult*

La première étape consiste à créer une unité qui constituera le noyau central de tous les composants. Pour ce faire, utilisez l'Expert composant comme précédemment et nommez le fichier `Tmultiply`. La Figure 13.5 montre comment vous devez remplir les champs de l'Expert pour créer `TMult`.

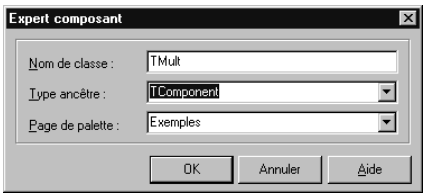


Figure 13.5
Les paramètres de l'Expert composant pour TMult.

Le code généré comprend un noyau pour la déclaration de classe de `TMult` et une procédure pour enregistrer la fonction. Pour développer les propriétés, la méthode et l'événement, vous devez modifier la définition de classe et fournir les procédures et fonctions correspondantes. Double-cliquez sur l'unité `TMultiply` dans le Gestionnaire de paquets pour voir apparaître le code source.

Ajouter des propriétés à *TMult*

`TMult` contient trois propriétés : `Val1`, `Val2` et `Res`. `Val1` et `Val2` sont disponibles lors de la conception et de l'exécution, tandis que `Res` ne l'est que lors de l'exécution. Comme chaque propriété contient des données, vous devez définir dans la classe `TMult` des variables qui contiendront ces dernières. Les utilisateurs n'accèdent aux variables que par un appel spécialisé. Par conséquent, on déclare les variables contenant les données des trois propriétés dans la partie `Private` de la définition de classe, ce qui signifie que seules les fonctions et les procédures de la classe pourront accéder aux données. Par convention, les noms des variables commencent par `F`, suivi du nom de la propriété. Dans notre exemple, toutes les propriétés sont des entiers, et leurs variables associées sont donc déclarées de type `integer`. La déclaration de classe est la suivante :

```
type
  TMult = class(TComponent)
Private
   FVal1 : integer;
   FVal2 : integer;
   FRes  : integer;
Protected
Public
Published
end;
```

Vous devez maintenant déclarer les propriétés elles-mêmes. Utilisez le mot clé `property` dans la définition de classe. La définition de propriété peut apparaître généralement en deux endroits. Si une propriété est accessible lors de la conception, elle doit être déclarée dans la section `Published` de la déclaration. Si elle n'est disponible que lors de l'exécution, elle est placée dans la section `public`. Dans notre exemple, les propriétés sont stockées sous forme de types de données simples et on n'effectue pas d'action particulière lorsque les données sont lues ou

écrites. On peut, par conséquent, utiliser un accès direct pour lire et écrire la propriété. Avec l'*accès direct*, vous indiquez à Delphi de modifier ou de renvoyer les données d'une variable lorsqu'une propriété est écrite ou lue. Les méthodes `read` et `write` définissent les variables. Voici les définitions de nos trois propriétés :

```
type
  TMult = class(TComponent)
Private
   FVal1 : integer;
   FVal2 : integer;
   FRes  : integer;
protected
public
   Property Res:integer read FRes;     {Propriété pour obtenir le résultat}
published
   property Val1:integer read FVal1 Write FVal1 default 1; {Opérande 1}
   property Val2:integer read FVal2 Write FVal2 default 1; {Opérande 2}
end;  {TMult}
```

Comme la propriété `Res` est en lecture seule, vous n'avez pas besoin d'une méthode à accès direct pour écrire dans la variable `FRes`.

`Val1` et `Val2` sont définies à 1 par défaut, ce qui peut prêter à confusion. La valeur par défaut d'une propriété est en fait définie dans une autre étape de la création du composant, lorsqu'un ajoute un constructeur. Delphi l'utilise dans la ligne de propriété pour déterminer s'il doit l'enregistrer lorsqu'un utilisateur enregistre un fichier de fiche. Quand ce dernier ajoute ce composant à une fiche et laisse `Val1` à 1, la valeur n'est pas enregistrée dans le fichier .dfm. Si la valeur est autre, elle est enregistrée.

Vous avez déclaré les propriétés du composant. Si vous installez le composant maintenant, `Val1` et `Val2` apparaîtront dans l'onglet Propriétés. Il vous reste quelques étapes avant d'obtenir un composant fonctionnel.

Ajouter le constructeur

Un constructeur est appelé lorsqu'une classe est créée. C'est lui qui est souvent chargé de l'allocation de mémoire dynamique ou de la collecte des ressources dont a besoin une classe. Il définit aussi les valeurs par défaut des variables d'une classe. Lorsqu'un composant est ajouté à une fiche, lors de la conception ou de l'exécution, le constructeur est appelé. Pour déclarer ce dernier dans la définition de classe, ajoutez une ligne `constructor` dans la partie `public` de la déclaration de classe. Par convention, on utilise `Create` comme nom de la procédure du constructeur, comme on peut le voir dans cet exemple :

```
{…}
public
```

```
constructor Create(AOwner : TComponent); override;{Constructeur principal }
{…}
```

On passe un paramètre au constructeur : le composant auquel appartient le constructeur. Ce n'est pas le même cas de figure que pour la propriété ancêtre. Vous devez spécifier que vous souhaitez ignorer le constructeur par défaut de la classe de l'ancêtre, qui est `TComponent`. Dans la portion implémentation de l'unité, vous ajoutez le code pour le constructeur.

```
constructor TMult.Create(AOwner: TComponent);
BEGIN
  inherited Create(AOwner); {appel du constructeur pour la classe du parent}
  FVal1 := 1;                          {Défaut pour valeur 1 }
  FVal2 := 1;                          {Défaut pour valeur 2 }
END; {End constructor}
```

Pour qu'une construction spécifique à un parent soit effectuée, vous devez commencer par appeler la procédure `Create` héritée. En l'occurrence, la seule étape supplémentaire consiste à définir des valeurs par défaut pour `Val1` et `Val2` qui correspondent à la section des valeurs par défaut des déclarations de propriétés.

Ajouter une méthode

Une méthode est plus facile à implémenter qu'une propriété. Pour en déclarer une, placez une procédure ou une fonction dans la partie `public` de la définition de classe et écrivez la fonction ou procédure associée. En l'occurrence, vous ajoutez la méthode `DoMult` :

```
{…}

public
 procedure DoMult;          {Méthode pour multiplier}

{…}

procedure TMult.DoMult;
Begin
  FRes := FVal1 * FVal2
End;
```

Votre composant fonctionne maintenant. L'utilisateur peut l'ajouter à une fiche et définir des valeurs pour `Val1` et `Val2` lors de la conception et de cette dernière étape. Lors de l'exécution, la méthode `DoMult` peut être appelée pour effectuer la multiplication.

Ajouter un événement

Un événement permet à l'utilisateur d'exécuter un code spécialisé lorsque que quelque chose arrive. Pour votre composant, vous pouvez ajouter un événement qui est déclenché lorsque `Val1` ou `Val2` est plus grand que 100 et que `DoMult` est exécuté ou, par exemple, modifier le code pour que, dans une telle éventualité, `Res` reste inchangé.

En Delphi, un événement est une propriété spécialisée, c'est-à-dire un pointeur vers une fonction. Tout ce qui s'applique aux propriétés s'applique dans ce cas aux fonctions.

Le composant doit fournir une dernière information : le moment où il faut appeler le Gestionnaire d'événements de l'utilisateur. Rien de plus simple. Dès qu'il est temps de déclencher un événement, il suffit de voir si l'utilisateur a défini un Gestionnaire d'événements. Si c'est le cas, on appelle l'événement. Vous devez ajouter les déclarations suivantes à la définition de classe :

```
{…}
Private
   FTooBig : TNotifyEvent;
   {…}
published
   Property OnTooBig:TNotifyEvent read FTooBig write FTooBig;  {Evénement}
   {…}
end;  {TMult}
 {…}
```

Le type `TNotifyEvent` est utilisé pour définir `FTooBig` et `OnTooBig` est un type de pointeur de fonction générique qui passe un paramètre de type `component`, `Self` en général. La dernière étape consiste à modifier la procédure `Tmult.DoMult` pour qu'elle appelle le Gestionnaire d'événements si l'un des nombres est trop grand. Avant d'appeler ce gestionnaire, vous devez regarder si un événement a été défini. Pour ce faire, utilisez la fonction `assigned`, qui renvoie `True` si un événement est défini pour le Gestionnaire d'événements et `False` sinon.

Le Listing 13.4 montre le code du composant.

Listing 13.4 : : Le composant TMult au complet

```
unit TMultiply;

interface

uses
 SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
 Forms, Dialogs, StdCtrls;

```

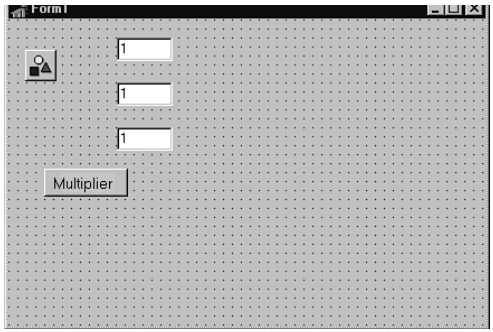
```
type
 TMult = class(TComponent)
Private
  FTooBig : TNotifyEvent;
  FVal1 : integer;
  FVal2 : integer;
  FRes  : integer;
protected
public
  {constructeur principal }
  constructor Create(AOwner : TComponent); override;
  {méthode pour multiplier }
  procedure DoMult;
  {propriété pour obtenir le résultat }
  Property Res:integer read FRes;
published
  property Val1:integer read FVal1 Write FVal1 default 1; {Opérande 1}
  property Val2:integer read FVal2 Write FVal2 default 1; {Opérande 2}
  Property OnTooBig:TNotifyEvent read FTooBig write FTooBig;  {événement}
end;  {TMult}

procedure Register;

implementation

constructor TMult.Create(AOwner: TComponent);
BEGIN
 inherited Create(AOwner);
 FVal1 := 1;
 FVal2 := 1;
End;

procedure TMult.DoMult;
Begin
if (Val1 < 100) and (Val2  < 100) then
 FRes := FVal1 * FVal2
else
 if assigned(FTooBig) then OnTooBig(Self);
End;

procedure Register;
```

```
begin
 RegisterComponents('Samples', [TMult]);
end;

end.
```

Tester le composant

Vous pouvez installer ce composant en suivant la procédure décrite plus tôt dans cette section. Une fois installé, vous pouvez le tester en créant une application qui en fait usage. La Figure 13.6 indique la disposition des composants pour l'application de test, et le Listing 13.5 montre le code de cette application.

Figure 13.6
La disposition des composants pour l'application de test de TMult.



Listing 13.5 : Unité principale pour le test de TMult

```
unit main;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs,
  StdCtrls, TMultiply;

type
  TForm1 = class(TForm)
    Mult1: TMult;
    EdVal1: TEdit;
    EdVal2: TEdit;
```

```
    EdResult: TEdit;
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Mult1TooBig(Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
begin
   { On définit les propriétés }
   Mult1.Val1 := StrToInt(EdVal1.Text);
   Mult1.Val2 := StrToInt(EdVal2.Text);
   { On exécute la méthode }
   Mult1.DoMult;
   {On capture le résultat }
   EdResult.Text := IntToStr(Mult1.Res);
end;

procedure TForm1.Mult1TooBig(Sender: TObject);
begin
   Application.MessageBox('Petit bras le composant',
                          'Le composant trouve le nombre trop grand',
                           MB_OK);
end;
end.
```

Analyse *L'application de test consiste en trois boîtes d'édition et en un bouton. Un utilisateur peut placer des nombres dans les deux premières. Lorsqu'il clique sur le bouton, les nombres sont multipliés à l'aide du composant et le résultat est placé dans la troisième. Vous pouvez ajouter*

un Gestionnaire d'événements qui affiche une boîte de dialogue indiquant qu'au moins un des nombres est trop grand, dans le cas où l'événement `OnTooBig` *s'est déclenché*.

Comme le montre la Figure 13.7, le Gestionnaire d'événements a été appelé lorsqu'un nombre supérieur à 100 a été multiplié dans le composant `TMult`.

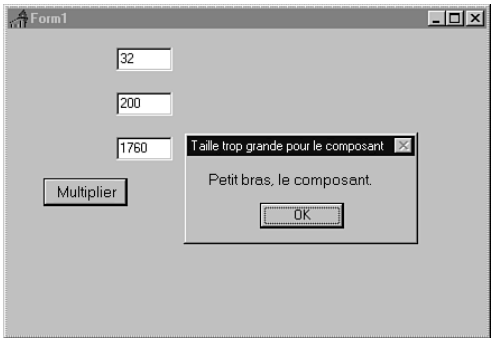


Figure 13.7
Le Gestionnaire d'événements de l'application de test pour TMult.

Lire et modifier des valeurs de propriété avec des procédures

Pour obtenir le produit de deux nombres, le composant `TMult` s'appuie sur un processus peu élégant en deux étapes. Lorsque la valeur de `Val1` ou `Val2` est modifiée, `Res` doit être recalculé automatiquement. Pour ce faire, vous pouvez appeler une procédure dès que la propriété est modifiée, ou encore une fonction qui renvoie la valeur de la propriété dès que celle-ci est lue.

Nouveau

Une méthode d'accès *est le processus d'appel d'une procédure ou d'une fonction lorsqu'on accède à une propriété. Pour en utiliser une, vous remplacez le nom de la variable de stockage direct par celui de la fonction utilisée pour manipuler les données de la déclaration de propriété. Pour implémenter une méthode d'accès dans la méthode* `TMult`*, vous devez apporter les modifications suivantes à la déclaration de classe :*

```
{...}
type
  TMult = class(TComponent)
Private
   FTooBig : TNotifyEvent;
   FVal1 : integer;
   FVal2 : integer;
   FRes  : integer;
   {**********  On déplace DoMult dans la zone Private *******}
```

```
    procedure DoMult;
    {**********  On ajoute la définition de SetVal1 et SetVal2 *******}

    procedure SetVal1(InVal : Integer);   {Pour définir Value1}
    procedure SetVal2(InVal : Integer);   {Pour définir Value2}
protected
public
    {Propriété pour obtenir le résultat }
    Property Res:integer read FRes;
   constructor Create(AOwner : TComponent); override;{constructeur principal }

published
    {**********  méthodes d'accès set *******}
    property Val1:integer read FVal1 Write SetVal1 default 1; {Operand 1}
    property Val2:integer read FVal2 Write SetVal2 default 1; {Operand 2}
    Property OnTooBig:TNotifyEvent read FTooBig write FTooBig;  {Event}
end;  {TMult}
{...}

procedure TMult.SetVal1(InVal : Integer);
Begin
  FVal1 := InVal;
  DoMult;
End;
procedure TMult.SetVal2(InVal : Integer);
Begin
  FVal2 := InVal;
  DoMult;
End;
{...}
end.
```

Dans le programme de test, vous n'avez plus besoin d'appeler la méthode `DoMult`. En fait, si vous tentez de le faire, l'application ne se compile pas car la méthode a été déplacée dans la partie privée. Les fonctionnalités demeurent inchangées pour le reste.

Modifier un composant déjà existant : *TButClock*

`TMult` nous a permis d'aborder de nombreux concepts liés à la création d'un composant, mais pas de voir comment dériver un composant d'un déjà existant. L'un des grands avantages de la programmation orientée objet est qu'un objet peut être dérivé d'une classe ancêtre. Ainsi, par exemple, si vous souhaitez créer un composant qui soit un bouton vert, vous n'êtes pas obli-

gé d'écrire tout le code qui crée un bouton et prévoit toutes les interactions possibles avec celui-ci. Un bouton générique existe déjà.

Grâce à l'héritage, vous pouvez dériver un nouvelle classe qui bénéficie de toutes les fonctionnalités de sa classe parent, auxquelles peuvent s'ajouter des améliorations ou des personnalisations. Dans l'exemple qui suit, nous allons créer un composant appelé `TButClock`. Il se comporte comme un bouton, à cela près que la propriété `Caption` est modifiée automatiquement pour contenir l'heure courante.

Pour construire un composant à partir d'un déjà existant, vous utilisez l'Expert composant pour créer son noyau et ajouter d'éventuelles nouvelles fonctionnalités. Pour cet exemple, on utilisera un thread qui sera placé dans une boucle jusqu'à destruction du composant. Le thread est en sommeil une seconde, puis appelle une fonction de callback dans le composant pour actualiser la caption avec l'heure courante. Le code du composant `TButClock` figure dans le Listing 13.6.

Listing 13.6 : Le composant TButClock

```
unit unitTBC;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs,
  StdCtrls,extctrls;
type
 {un type Callback pour que le thread puisse actualiser l'horloge }
  TCaptionCallbackProc = procedure(X : String) of object;

 {Objet Thread }
  TUpdateClock = class(TThread)
  private
    procedure UpdateCaption;
  protected
    procedure Execute; override;
  public
    UpdateClockProc : TCaptionCallbackProc;
  end;

  {Composant principal }
  TButClock = class(TButton)
  private
    { Déclarations privées }
```

```
    MainThread  : TUpdateClock;
    procedure UpdateCaption(X : String);
  protected
    { Déclarations protégées }
  public
    { Déclarations publiques }
    constructor Create(AOwner : TComponent);override;
    destructor  destroy; override;
  published
    { Déclarations publiées }
  end;

procedure Register;

implementation

procedure TUpdateClock.UpdateCaption;
{Cette routine appelle UpdateClockProc en lui fournissant }
{l'heure correcte }
begin
   UpdateClockProc(TimeToStr(Now));
end;

procedure TUpdateClock.Execute;
{On boucle jusqu'à ce qu'on nous dise de terminer, puis on sort }
begin
 while (not Terminated) do
 begin
  Synchronize(UpdateCaption);
  Sleep(1000);
 end
end;

constructor TButClock.Create(AOwner : TComponent);
begin
    inherited Create(AOwner);
    {on crée le thread en mode suspendu }
    MainThread := TUpdateClock.Create(True);
    {on définit le pointeur de callback }
    MainThread.UpdateClockProc := UpdateCaption;
    {On sort le thread du mode suspendu }
```

```
    MainThread.Resume;
end;

destructor TButClock.Destroy;
{Appelé lors de la destruction du composant }
begin
   { on dit au thread de s'achever }
   MainThread.Terminate;
   {On attend la fin }
   MainThread.WaitFor;
   { Nettoyage de TButClock}
   inherited Destroy;
end;

procedure TButClock.UpdateCaption(X : String);
{On modifie le Caption lorsqu'on nous dit de le faire }
begin
   Caption := X;
end;

procedure Register;
begin
  RegisterComponents('Samples', [TButClock]);
end;

end.
```

Analyse

L'analyse de ce code figure dans les parties qui suivent.

Le constructeur

Le constructeur commence par exécuter tout le code nécessaire provenant de son parent. En l'occurrence, l'objet est un descendant de `TButton`, il est donc important que le bouton procède aux allocations ou initialisations indispensables à son bon fonctionnement. Une fois que vous avez appelé le constructeur hérité, vous créez une instance de la classe de thread `TUpdateClock` en appelant sa méthode `create`. Celle-ci est passée comme `True` pour indiquer que le thread doit être suspendu lors de sa création. Une fois ce dernier créé, vous devez procéder comme suit :

1. Affectez la procédure `UpdateCaption` à la propriété `UpdateClocProc` du thread. Vous indiquez ainsi au thread ce qu'il doit faire lorsqu'il est temps d'actualiser le libellé du bouton.

2. Sortez le thread du mode suspendu en appelant `Resume`, comme le montre le code ci-après :

```
constructor TButClock.Create(AOwner : TComponent);
begin
    inherited Create(AOwner);
    {on crée le thread en mode suspendu }
    MainThread := TUpdateClock.Create(True);
    {on définit le pointeur de callback }
    MainThread.UpdateClockProc := UpdateCaption;
    {on sort le thread du mode suspendu }
    MainThread.Resume;
end;
```

Le destructeur

Pour le destructeur, vous devez ordonner au thread de s'achever, puis attendre qu'il le soit avant de pouvoir détruire sans danger le composant `TButClock`, comme le montre le code ci-après :

```
destructor TButClock.Destroy;
{appelé lors de la destruction du composant }
begin
   { indique au thread de s'achever }
   MainThread.Terminate;
   { on attend la fin }
   MainThread.WaitFor;
   { Nettoyage de TButClock}
   inherited Destroy;
end;
```

La procédure *UpdateCaption*

Le thread appelle `UpdateCaption` et lui passe `Time` pour actualiser le libellé, comme le montre le code ci-après (vous auriez tout aussi bien pu placer la logique liée à l'heure dans cette procédure) :

```
procedure TButClock.UpdateCaption(X : String);
{On modifie le Caption lorsqu'on nous dit de le faire }
begin
   Caption := X;
end;
```

La procédure *Register*

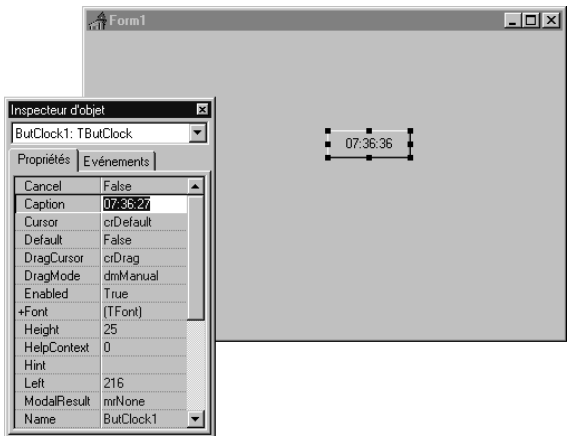
Vous achevez le composant avec la procédure `Register`, qui déclare que le composant `TButClock` doit être placé sur la page Exemples de la VCL.

```
procedure Register;
begin
  RegisterComponents('Samples', [TButClock]);
end;
end.
```

TButClock

Une fois que vous avez construit votre composant, il est prêt à l'emploi. Lorsque vous l'ajoutez à une forme lors de la conception, il donne l'heure avant même que le programme ne soit compilé. Vous n'avez pas modifié la fonctionnalité de la propriété `Caption` pour qu'elle puisse être lue : elle indique l'heure courante. Vous n'avez pas empêché l'utilisateur d'écrire dans le champ `Caption`. Tout ce qu'il écrit dans cette propriété sera remplacé par l'heure courante la prochaine fois que le thread inscrira la nouvelle heure. La Figure 13.8 montre une application utilisant le composant bouton horloge.

Figure 13.8
Le bouton horloge.



Déclarer un nouvel événement : *UserPlot*

Lorsque vous avez travaillé avec `TMult`, vous avez vu comment ajouter un événement à un composant. C'était cependant un cas particulier. Vous avez déclaré la propriété comme un type `TNotifyEvent`. Pour passer d'autres paramètres vers ou en provenance d'un événement, Del-

phi a déclaré `TNotify` comme un pointeur vers une fonction à laquelle on transmet comme paramètre un `TObject`. Si un événement doit utiliser d'autres paramètres, vous pouvez déclarer le type correspondant. L'exemple qui suit montre comment y parvenir. Il permet de créer un événement qui passe un nombre réel au Gestionnaire d'événements et renvoie un nombre réel différent en utilisant un paramètre `var`.

Selon le type d'applications que vous concevez, le composant qui suit peut s'avérer utile. On est souvent amené à tracer une fonction mathématique. De nombreux composants vous permettent de créer un graphique en fournissant un ensemble de points, mais très peu vous permettent de fournir simplement la fonction à tracer. Ce composant sert à cela. Un événement appelé `OnUserFunc` est défini. Il passe une valeur X et attend qu'une valeur Y soir renvoyée. Les facteurs d'intervalle et d'échelle sont définis comme propriétés. Ainsi, si vous souhaitez tracer la fonction Y=X×2, vous ajoutez le composant à votre forme et le code ci-après à l'événement `OnUserFunc` :

```
procedure TForm1.FuncGraph1UserFunc(X: Real; var Y: Real);
begin
  Y := X * X;
end;
```

Le composant `TFuncGraph` gère toutes les mises à l'échelle et transforme les coordonnées. Vous n'avez en fait à taper qu'une seule ligne de code. Les exemples précédents vous ont montré comment en implémenter la plus grande partie. Le code complet de `TFuncGraph` figure Listing 13.7. Les sections qui suivent mettent l'accent sur la méthode permettant de créer un nouveau type d'événement et sur la création d'un événement basé sur celui-ci.

Listing 13.7 : Le composant TFuncGraph

```
unit PlotChart;

interface

uses
 Windows, Messages, SysUtils, Classes, Graphics, Controls,
 Forms, Dialogs;

type
 TUserPlotFunc = procedure(X : real ; var Y : real) of object;
 TFuncGraph = class(TGraphicControl)
 private
    { Déclarations privées }
   FRangeMinX : integer;
   FRangeMaxX : integer;
   FRangeMinY : integer;
```

```
    FRangeMaxY : integer;
    FUserFunc  : TUserPlotFunc;
  protected
    { Déclarations protégées }
    procedure paint; override;
  public
    { Déclarations publiques }
    constructor Create(Aowner : TComponent); override;
  published
    { Déclarations publiées }
    property RangeMinX : integer read FRangeMinX write FRangeMinX;
    property RangeMaxX : integer read FRangeMaxX write FRangeMaxX;
    property RangeMinY : integer read FRangeMinY write FRangeMinY;
    property RangeMaxY : integer read FRangeMaxY write FRangeMaxY;
    property OnUserFunc : TUserPlotFunc read FUserFunc write FUserFunc;
    property Width default 50;
    property Height default 50;
  end;

procedure Register;

implementation

constructor TFuncGraph.Create(Aowner : TComponent);
begin
{on définit une largeur, une hauteur et un intervalle par défaut }
   inherited Create(AOwner);
   Height :=  50;
   Width  :=  50;
   FRangeMaxX := 1;
   FRangeMaxY := 1;
end;

procedure TFuncGraph.Paint;
var
  X,Y   : integer; { pixels réels}
  RX,RY : real;    {coordonnées utilisateur }

begin
  inherited Paint;
  Canvas.Rectangle(0,0,Width,Height);
```

```
 For X := 1 to Width do
  begin
    {on convertit X en X utilisateur }
    {Note : la largeur ne peut être 0}
    RX := FRangeMinX + (((FRangeMaxX - FRangeMinX)/Width)*X);

    {Si l'utilisateur a affecté une fonction de traçage }
    {on appelle cette fonction, sinon on affecte RY = 0           }
    if assigned(FUserFunc) then
      FUserFunc(RX,RY)
     else
      RY := 0;

    {On reconvertit RY en coordonnées de pixel }
    Y := round((1-((RY-FRangeMinY)/(FRangeMaxY-FRangeMinY)))* Height);
    if X = 1 then
      Canvas.MoveTo(X,Y)
     else
      Canvas.LineTo(X,Y);
  end;

end;

procedure Register;
begin
 RegisterComponents('Additional', [TFuncGraph]);
end;

end.
```

Analyse *L'analyse de ce listing figure dans les parties qui suivent.*

Créer un nouveau type d'événement

La fonctionnalité clé de ce composant est de permettre à l'utilisateur de définir une fonction arbitraire à tracer. Pour ce faire, vous implémentez un nouveau type d'événement, `TUser-PlocFunc`, dont la définition est :

```
TUserPlotFunc = procedure(X : real ; var Y : real) of object;
```

Ce type est déclaré dans la partie type de l'unité. Notez que `TUserPlotFunc` est une procédure, ce qui peut sembler bizarre dans une section type. Cela signifie que vous pouvez déclarer une variable qui est un pointeur vers une procédure prenant les arguments spécifiés dans la déclaration de type. Une fois le type déclaré, vous définissez une propriété publiée de `TUserPlotFunc` pour créer un événement utilisant les paramètres définis précédemment :

```
published
   property OnUserFunc : TUserPlotFunc read FUserFunc write FUserFunc;
```

Lorsque le composant est installé, un nouvel événement appelé `OnUserFunc` apparaît dans la liste. Si on double-clique dessus, Delphi crée une nouvelle procédure contenant les paramètres adéquats.

```
procedure TForm1.FuncGraph1UserFunc(X: Real; var Y: Real);
begin

end;
```

Appeler l'événement

Pour appeler le Gestionnaire d'événements à partir de votre composant, vous appelez la variable qui pointe sur la procédure et vous passez les paramètres adéquats. Assurez-vous qu'un événement valide est défini. Pour le savoir, appelez la fonction `assigned`. Voici un exemple d'appel à la fonction utilisateur :

```
if assigned(FUserFunc) then
       FUserFunc(RX,RY)
```

TFuncGraph

La Figure 13.9 montre la puissance potentielle d'un événement personnalisé. En ne tapant que huit lignes de code (voir Listing 13.8), vous pouvez tracer quatre fonctions mathématiques. Voilà ce qui s'appelle du développement rapide d'application.

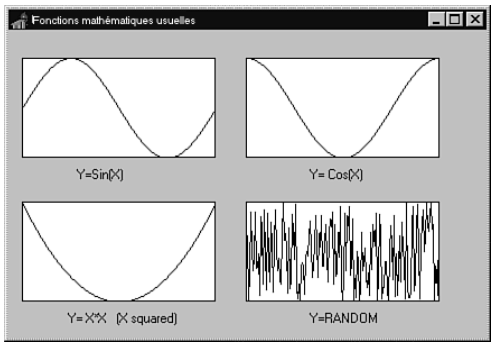


Figure 13.9
Utilisez le composant TFuncGraph *pour tracer quatre fonctions mathématiques.*

Listing 13.8 : Programme de test de TFuncGraph

```
unit unitPlotApp;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs,
  StdCtrls, PlotChart;

type
  TForm1 = class(TForm)
    FuncGraph4: TFuncGraph;
    FuncGraph1: TFuncGraph;
    FuncGraph2: TFuncGraph;
    FuncGraph3: TFuncGraph;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    procedure FuncGraph1UserFunc(X: Real; var Y: Real);
    procedure FuncGraph3UserFunc(X: Real; var Y: Real);
    procedure FuncGraph4UserFunc(X: Real; var Y: Real);
    procedure FuncGraph2UserFunc(X: Real; var Y: Real);
  private
    { Déclarations privées }
  public
```

```
    { Déclarations publiques }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FuncGraph1UserFunc(X: Real; var Y: Real);
begin
  Y := X;
end;

procedure TForm1.FuncGraph3UserFunc(X: Real; var Y: Real);
begin
  Y := Cos(X);
end;

procedure TForm1.FuncGraph4UserFunc(X: Real; var Y: Real);
begin
  Y := Sin(X);
end;

procedure TForm1.FuncGraph2UserFunc(X: Real; var Y: Real);
begin
  Y := sqrt(X);
end;

end.
```

Analyse

Cette application de test simple trace quatre fonctions, Y=X, Y=sin(X), Y=cos(X) et Y=sqrt(X). L'échelle pour les coordonnées Y ainsi que l'intervalle des X sont définies dans les propriétés de chacune des quatre coordonnées. Les composants `TLabel` *sont utilisés pour inscrire un titre sous chacune des courbes.*

Présentation d'ActiveX et de ses composants

Microsoft a d'abord lancé OLE (*Object Linking and Embedding*, ou liaison et incorporation d'objets) comme standard permettant à des objets de communiquer avec une application hôte. La spécification initiale avait pour but de permettre à une application telle qu'Excel d'incorporer une feuille de calcul dans n'importe quelle autre application prenant en charge le standard OLE. OLE 1.x était dépourvu de certaines fonctionnalités indispensables, si bien qu'une spécification OLE 2.0 a été mis en place, puis implémentée. OLE représentait les fondations d'une technologie permettant le partage d'objets génériques. Cette technologie s'est appelée COM (*Component Object Model* ou Modèle objet de composant). et a été utilisée pour créer une spécification pour les composants OCX. L'acronyme OLE s'est révélé réducteur puisque COM était loin de se cantonner à l'incorporation et à la liaison d'objets. Microsoft a alors développé le standard ActiveX actuel, ainsi que les composants ActiveX, qui sont les successeurs d'OLE et des OCX fondés sur l'architecture COM.

On peut se représenter COM comme le standard binaire de partage de composants entre deux morceaux de code. COM permet de séparer l'implémentation d'un objet des fonctions que cet objet effectue. Les fonctions qu'il effectue sont décrites dans ses interfaces. Une interface est une méthode d'accès à un ensemble de fonctions logiquement apparentées, que peut implémenter un objet. Chaque classe d'objet possède un identificateur (ID) de classe unique (`CLSID`) qui prend en charge un ensemble arbitraire d'interfaces. Toutes les classes doivent prendre en charge l'interface `IUnknown` qui peut être ensuite utilisée pour accéder aux interfaces qu'elle gèrent. Ceci s'effectue par le biais de la fonction `QueryInterface`, qui est toujours fournie dans l'interface `IUnknown`. Celle-ci permet à une application de demander à un objet s'il prend en charge les fonctions lui permettant d'effectuer telle ou telle tâche. L'objet répond alors par oui ou par non. Ce modèle objet est très puissant car il permet à une application de déterminer cela en phase d'exécution.

Un objet COM est implémenté par le biais de plusieurs méthodes. Il peut être compilé en une DLL ou un OCX s'exécutant dans le même espace de processus que l'application qui l'appelle. Il peut également être lancé dans son propre espace, sous forme d'exécutable compilé. Avec COM distribué (DCOM), l'objet peut s'exécuter sur une machine différente, n'importe où dans le monde. Les services système COM simplifient l'appel d'objets COM, même si le code d'implémentation se trouve dans un processus ou sur une machine différente. Les composants ActiveX sont des objets COM qui implémentent un ensemble d'interfaces de base permettant au composant d'être incorporé à des applications qui accueillent des composants ActiveX.

Avec Delphi, il est très simple de créer des composants ActiveX, même à partir d'un composant visuel Delphi déjà existant. L'application hôte peut alors manipuler les propriétés et répondre aux événements tout comme une application Delphi avec des composants visuels. Vous pouvez également ajouter de nouveaux événements, propriétés et méthodes au composant ActiveX pour lui donner des fonctionnalités supplémentaires.

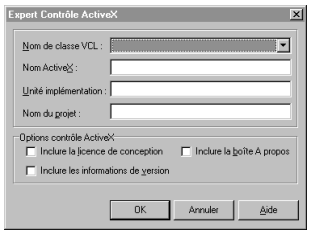
Convertir un composant visuel en un composant

ActiveX

La première étape pour créer un composant ActiveX consiste à créer une nouvelle bibliothèque ActiveX. Pour la créer, choisissez Fichier, Nouveau dans le menu, sélectionnez l'onglet ActiveX, puis choisissez Bibliothèque ActiveX. Vous créez ainsi un nouveau projet qui se compilera sous forme de fichier `.OCX` (le module qui stocke les composants ActiveX). Ensuite, choisissez Fichier, Nouveau dans le menu Delphi. Dans la boîte de dialogue qui apparaît alors, choisissez Contrôle ActiveX. L'assistant du même nom apparaît, c'est celui qui générera le code nécessaire à la création d'un contrôle ActiveX à partir d'un composant visuel déjà existant.

L'assistant a besoin de trois informations : le composant visuel sur lequel sera fondé le composant ActiveX, la classe du nouveau composant ActiveX et l'emplacement du futur fichier d'implémentation. D'autres options vous permettent d'utiliser des licences de conception, le contrôle des versions et une boîte A propos. Pour notre exemple, nous allons partir du composant visuel bouton horloge (`TButtonClock`) pour en faire un contrôle ActiveX. La Figure 13.10 montre ce à quoi doit ressembler l'Assistant.

Figure 13.10
L'Assistant contrôle ActiveX.



L'Assistant génère tout le code nécessaire à la compilation du composant en un composant ActiveX. Pour compiler le contrôle, il suffit de choisir Projet, Compiler dans le menu Delphi.

Pour ajouter des fonctionnalités à un contrôle ActiveX, vous pouvez employer deux méthodes. La première consiste à les ajouter au composant visuel sur lequel est basé le contrôle ActiveX et à le construire à nouveau. L'autre méthode consiste à ajouter directement les fonctionnalités au composant ActiveX. Le code source généré par l'Assistant contrôle ActiveX figure Listing 13.9.

Listing 13.9 : Le code source du contrôle ActiveX généré par l'assistant contrôle ActiveX à partir du composant bouton horloge

```
unit ActXClockImpl;

interface

uses
```

```
  Windows, ActiveX, Classes, Controls, Graphics, Menus, Forms,
  [ccc]StdCtrls,
  ComServ, StdVCL, AXCtrls, ActXClockPR_TLB, unitTBC;

type
  TActXClockX = class(TActiveXControl, IActXClockX)
  private
    { Déclarations privées }
    FDelphiControl: TButClock;
    FEvents: IActXClockXEvents;
  protected
    { Déclarations protégées }
    procedure InitializeControl; override;
    procedure EventSinkChanged(const EventSink: IUnknown); override;
    procedure DefinePropertyPages(DefinePropertyPage:
      [ccc]TDefinePropertyPage); override;
    function Get_Cancel: WordBool; safecall;
    function Get_Caption: WideString; safecall;
    function Get_Cursor: Smallint; safecall;
    function Get_Default: WordBool; safecall;
    function Get_DragCursor: Smallint; safecall;
    function Get_DragMode: TxDragMode; safecall;
    function Get_Enabled: WordBool; safecall;
    function Get_Font: Font; safecall;
    function Get_ModalResult: Integer; safecall;
    function Get_Visible: WordBool; safecall;
    procedure Click; safecall;
    procedure Set_Cancel(Value: WordBool); safecall;
    procedure Set_Caption(const Value: WideString); safecall;
    procedure Set_Cursor(Value: Smallint); safecall;
    procedure Set_Default(Value: WordBool); safecall;
    procedure Set_DragCursor(Value: Smallint); safecall;
    procedure Set_DragMode(Value: TxDragMode); safecall;
    procedure Set_Enabled(Value: WordBool); safecall;
    procedure Set_Font(const Value: Font); safecall;
    procedure Set_ModalResult(Value: Integer); safecall;
    procedure Set_Visible(Value: WordBool); safecall;
  end;

implementation

```

```
{ TActXClockX }

procedure TActXClockX.InitializeControl;
begin
  FDelphiControl := Control as TButClock;
end;

procedure TActXClockX.EventSinkChanged(const EventSink: IUnknown);
begin
  FEvents := EventSink as IActXClockXEvents;
end;

procedure TActXClockX.DefinePropertyPages(DefinePropertyPage:
  [ccc]TDefinePropertyPage);
begin
  { Définissez les pages de propriété ici. Celle(s)-ci sont définies en
appelant
    DefinePropertyPage avec l'id de classe de la page. Par exemple,
      DefinePropertyPage(Class_ActXClockXPage); }
end;

function TActXClockX.Get_Cancel: WordBool;
begin
  Result := FDelphiControl.Cancel;
end;

function TActXClockX.Get_Caption: WideString;
begin
  Result := WideString(FDelphiControl.Caption);
end;

function TActXClockX.Get_Cursor: Smallint;
begin
  Result := Smallint(FDelphiControl.Cursor);
end;

function TActXClockX.Get_Default: WordBool;
begin
  Result := FDelphiControl.Default;
end;
```

```
function TActXClockX.Get_DragCursor: Smallint;
begin
  Result := Smallint(FDelphiControl.DragCursor);
end;

function TActXClockX.Get_DragMode: TxDragMode;
begin
  Result := Ord(FDelphiControl.DragMode);
end;

function TActXClockX.Get_Enabled: WordBool;
begin
  Result := FDelphiControl.Enabled;
end;

function TActXClockX.Get_Font: Font;
begin
  GetOleFont(FDelphiControl.Font, Result);
end;

function TActXClockX.Get_ModalResult: Integer;
begin
  Result := Integer(FDelphiControl.ModalResult);
end;

function TActXClockX.Get_Visible: WordBool;
begin
  Result := FDelphiControl.Visible;
end;

procedure TActXClockX.Click;
begin

end;

procedure TActXClockX.Set_Cancel(Value: WordBool);
begin
  FDelphiControl.Cancel := Value;
end;

procedure TActXClockX.Set_Caption(const Value: WideString);
```

```
begin
  FDelphiControl.Caption := TCaption(Value);
end;

procedure TActXClockX.Set_Cursor(Value: Smallint);
begin
  FDelphiControl.Cursor := TCursor(Value);
end;

procedure TActXClockX.Set_Default(Value: WordBool);
begin
  FDelphiControl.Default := Value;
end;

procedure TActXClockX.Set_DragCursor(Value: Smallint);
begin
  FDelphiControl.DragCursor := TCursor(Value);
end;

procedure TActXClockX.Set_DragMode(Value: TxDragMode);
begin
  FDelphiControl.DragMode := TDragMode(Value);
end;

procedure TActXClockX.Set_Enabled(Value: WordBool);
begin
  FDelphiControl.Enabled := Value;
end;

procedure TActXClockX.Set_Font(const Value: Font);
begin
  SetOleFont(FDelphiControl.Font, Value);
end;

procedure TActXClockX.Set_ModalResult(Value: Integer);
begin
  FDelphiControl.ModalResult := TModalResult(Value);
end;

procedure TActXClockX.Set_Visible(Value: WordBool);
begin
```

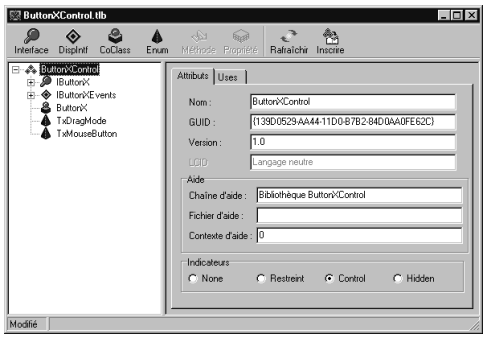
```
  FDelphiControl.Visible := Value;
end;

initialization
  TActiveXControlFactory.Create(
    ComServer,
    TActXClockX,
    TButClock,
    Class_ActXClockX,
    1,
    '',
    0);
end.
```

Analyse

Une nouvelle classe est générée (`TActXClockX`). Elle contient un composant visuel `TButClock` dans la section `private` de la définition de classe. Toutes les propriétés et méthodes du composant ActiveX sont définies comme des procédures et des fonctions dans sa déclaration. Ainsi, la propriété `Cursor` est implémentée avec la fonction `Get_Cursor` et la procédure `Set_Cursor`. Ces procédures sont appelées lorsque `Cursor` est définie ou lue. Leur implémentation est automatiquement générée par l'Assistant contrôle ActiveX.

En plus du fichier d'implémentation ActiveX, l'Assistant Control construit une bibliothèque de types, qui définit les interfaces et les propriétés du composant dans une bibliothèque ActiveX. Delphi propose un éditeur de bibliothèque de types vous permettant de modifier (et de consulter) les informations que celle-ci contient sur un contrôle ActiveX. Pour consulter cette bibliothèque, choisissez Voir, Bibliothèque de types, et vous pourrez alors voir quels contrôles, interfaces et pages de propriétés se trouvent dans le projet, ainsi que leurs propriétés, événements et méthodes (voir Figure 13.11).

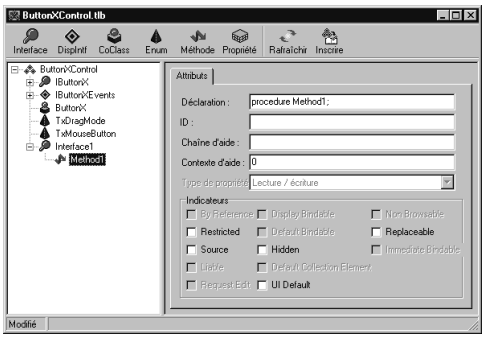
Figure 13.11
La bibliothèque de types du composant ActiveX bouton horloge.



Ajouter directement une méthode dans un composant ActiveX

Il est également très facile d'ajouter des propriétés, des événements et des méthodes directement dans un contrôle ActiveX. Pour cela, nous allons ajouter une nouvelle méthode, `MakeBold`, qui fera passer en gras le texte du libellé. Pour l'ajouter, choisissez Editer, Ajouter à l'interface dans le menu Delphi. La boîte de dialogue Ajout à l'interface apparaît alors. Assurez-vous que Interface est définie comme `Propriétés/méthodes`, et entrez `procedure MakeBold;` pour la déclaration (voir Figure 13.12).

Figure 13.12
Ajouter une méthode au composant.



Vous avez ainsi effectué trois tâches : la méthode `MakeBold` a été ajoutée à la définition d'interface dans la bibliothèque de types et à la définition de classe, et un squelette de la procédure `MakeBold` a été créé. Le voici :

```
procedure TActXClockX.MakeBold;
begin

end;
```

Il vous incombe alors de compléter ce code en ajoutant celui qui modifie la police. La procédure finale est la suivante :

```
procedure TActXClockX.MakeBold;
begin
   FDelphiControl.Font.Style := [fsBold];
end;
```

Le `FDelphiControl` référencé dans cette procédure est une instance du composant `TButClock`, qui est encapsulé dans le contrôle ActiveX. Lorsque la méthode `MakeBold` est appelée sur le composant ActiveX `TActXClockX`, la procédure affecte `fsBold` à la propriété `Font.Style` dans

le composant encapsulé. Vous pouvez voir la déclaration de `FDelphiControl` dans la définition de classe.

Lorsque vous compilez le projet, vous obtenez un OCX contenant l'implémentation du composant ActiveX. Les composants ActiveX doivent être enregistrés dans un système avant utilisation. La bibliothèque ActiveX s'auto-enregistre si l'application hôte peut appeler la procédure d'enregistrement. Il est également possible d'enregistrer le composant à partir de l'EDI Delphi en choisissant Exécuter, Recenser Serveur ActiveX. Ici, sauvegardez-le immédiatement à l'aide de cette option de menu.

Vous pouvez également utiliser Dé-recenser serveur ActiveX pour désinstaller un contrôle de votre machine. Le moyen le plus simple pour le tester consiste à utiliser la commande Déploiement Web pour que Delphi génère une page Web de test. Nous allons compliquer un peu ce principe dans la partie suivante en incorporant le composant dans une page Web contenant un script qui permettra à ce composant d'interagir avec d'autres composants de la page.

Composant ActiveX dans une page Web contenant un script

L'aspect le plus séduisant des ActiveX est qu'ils sont indépendants du langage et de l'application. Dans l'exemple qui va suivre, nous allons créer une page Web en comportant deux : le désormais célèbre bouton horloge et le bouton poussoir Microsoft standard. On ajoutera du VBScript à la page pour que, lorsque l'utilisateur clique sur le bouton poussoir, la police du bouton horloge passe en gras.

Le code HTML correspondant à cette page figure Listing 13.10.

Listing 13.10 : Code HTML pour créer une page contenant des composants ActiveX

```
<HTML>
<HEAD>
<TITLE>Des composants ActiveX marchant main dans la main ! </TITLE>
</HEAD>
<BODY>
<H2> Des composants ActiveX marchant main dans la main !</H2>
<HR>
<CENTER><Table Border=1>
<TR><TD ALIGN=CENTER>
 <OBJECT ID="PushForBold" WIDTH=203 HEIGHT=32
 CLASSID="CLSID:D7053240-CE69-11CD-A777-00DD01143C57">
    <PARAM NAME="VariousPropertyBits" VALUE="268435483">
    <PARAM NAME="Caption" VALUE="Cliquez ici pour enrichir l'horloge ">
    <PARAM NAME="Size" VALUE="4313;678">
</OBJECT>
```

```
</TD></TR>
<TR><TH>Appuyez sur le bouton ci-avant pour faire passer l'horloge en gras<
TH></TR>
<TR><TD ALIGN=CENTER>
 <OBJECT ID="ClockButton" WIDTH=75 HEIGHT=25
     CLASSID="CLSID:C8EE0B43-8C8F-11D0-9FB3-444553540000">
    </OBJECT>
</TD></TR></TABLE></CENTER><HR>
L'horloge et le bouton sont des composants ActiveX. Le reste est du HTML on
ne peut plus classique

<SCRIPT LANGUAGE="VBScript">
   Sub PushForBold_Click()
      call ClockButton.MakeBold()
   end sub
</SCRIPT>
</BODY>
</HTML>
```

Analyse *Chacun des composants ActiveX est marqué par une balise `<OBJECT>` qui inclut un ID le référençant dans la page. Dans cette balise, on spécifie des informations concernant le composant. La plus cruciale se trouve dans la propriété `CLASSID`. C'est là qu'on précise quel composant ActiveX sera placé sur la page pour cet objet. L'ID de classe correspond au `CoClass GUID` affiché dans la bibliothèque de types de Delphi. Les autres paramètres de la balise `<OBJECT>` spécifient d'éventuels paramètres d'initialisation pour l'objet. Ainsi, sur le bouton de commande, on saisit le libellé `Cliquez ici pour enrichir l'horloge`. Il faut aussi spécifier un Gestionnaire d'événements en faisant en sorte que le VBScript appelle la méthode `MakeBold` lorsqu'on appuie sur le bouton poussoir. Vous auriez pu tout aussi bien utiliser JavaScript ou d'autres langages de script.*

Récapitulatif

Delphi est un outil de choix pour l'écriture d'applications. Il l'est également pour la création de composants utilisables avec Delphi et d'autres applications. Nous avons vu comment fonctionnent les composants et ce qui est nécessaire pour créer, installer et utiliser les vôtres composants. Une des fonctionnalités les plus puissantes du modèle objet de Delphi tient au fait qu'il est orienté objet, ce qui signifie que vous pouvez prendre un composant déjà existant pour l'améliorer. Nous avons également vu comment créer et utiliser des composants ActiveX. Ceux-ci peuvent être exploités avec Delphi et un grand nombre d'autres applications. Retenez que si vous écrivez votre code sous forme de composants réutilisables, quand vous devrez

effectuer une tâche similaire, celui correspondant existera déjà et sera prêt à l'emploi. Le temps ainsi gagné est incalculable.

Atelier

L'atelier vous donne deux façons de vérifier que vous avez correctement assimilé le contenu de cette partie. La section Questionnaire vous pose des questions, dont vous trouverez les réponses en consultant l'Annexe A, et les Exercices vous permettent de mettre en pratique ce que vous venez d'apprendre. Tentez, dans la mesure du possible, de bien consulter chacune des trois sections avant de passer à la suite.

Questions - Réponses

Q Puis-je créer un composant visuel sans hériter d'une classe d'ancêtre ?

R Non. Tout composant doit être un descendant d'une classe, même de la plus fondamentale (`TComponent`).

Q Puis-je intégrer une aide dans mon composant ?

R Oui. Delphi permet de nombreuses connexions vers l'aide. Consultez le guide du concepteur de composants pour plus de détails.

Q Puis-je créer des pages de propriétés personnalisées pour mes composants ActiveX ?

R Oui. Vous pouvez choisir Fichier, Nouveau pour accéder à la boîte de dialogue Nouveaux éléments, puis sélectionner l'onglet ActiveX avant de choisir "page de propriété". Cette page peut alors être liée au composant.

Q Puis-je convertir un composant ActiveX en un composant visuel ?

R Oui. Delphi vous permet de créer un emballage de composant visuel autour d'un composant ActiveX déjà existant.

Questionnaire

1. Les propriétés des composants ne sont-elles que des variables, ou est-il possible de lier du code aux propriétés d'un composant ?
2. Quelle est la différence entre les propriétés disponibles lors du développement et celles qui ne le sont que lors de l'exécution ?
3. Quelle est la différence entre publié et public ?
4. En quoi diffère la définition d'un événement de celle d'une propriété ?
5. A quoi servent l'ID de classe, la bibliothèque de types et l'interface ?
6. Peut-on utiliser des composants ActiveX ailleurs que dans Delphi ?

Exercices

1. Ajoutez une propriété au composant `TButClock` permettant à l'utilisateur de spécifier une heure de sonnerie. Ajoutez également un événement `OnAlarm` qui est appelé lorsque l'heure spécifiée survient.
2. Ajoutez la fonctionnalité décrite ci-avant au composant bouton horloge ActiveX.
3. Transformez le composant `TPanel` en un composant ActiveX, et améliorez-le en écrivant le code pour qu'il puisse afficher un message qu'une application aura déterminé en modifiant une propriété.

14

Créer des applications Internet avec Delphi

Depuis quelques années, le mot "Internet" est sur toutes les lèvres. La puissance de l'Internet est due en partie aux protocoles (langages utilisés par les applications de réseau pour communiquer) permettant à n'importe quel utilisateur d'accéder à des informations et à des applications situées n'importe où dans le monde. Ces mêmes concepts permettent à des sociétés de diffuser des informations internes à tous leurs employés en utilisant des intranets. Un intranet est un réseau privé dont les ressources sont similaires à celles de l'Internet, mais qui ne sont accessibles qu'à des personnes dont l'accès est autorisé.

Nombreux sont ceux pour qui "Internet" et "Web" sont une seule et même chose. Ce n'est pas le cas. Le World Wide Web (WWW ou Web) n'est qu'une des applications, ou ensemble de protocoles, utilisant l'Internet comme mécanisme de livraison. L'Internet est le réseau physique et logique qui interconnecte toutes les machines qui y sont reliées. Le protocole de réseau par le biais duquel communiquent les machines de l'Internet est TCP/IP (*Transmission Control Protocol/Internet Protocol*). Les serveurs et browsers Web communiquent au moyen de protocoles de niveau supérieur — principalement HTTP et FTP — pour transmettre les informations entre le client et le serveur.

Il est maintenant possible d'utiliser n'importe quel protocole s'appuyant sur TCP/IP afin d'exploiter l'Internet pour communiquer. De même, le protocole HTPP peut-être utilisé pour permettre à une application de communiquer avec n'importe quelle autre et non uniquement vers le Web. Vous pouvez ainsi développer un jeu de course automobile dans lequel deux conducteurs s'affrontent, les données étant partagées entre les deux à l'aide de HTTP.

Lors de cette journée, vous découvrirez les concepts de base du Web et la façon de produire, avec Delphi, des applications Web client et serveur robustes. Delphi est un outil très puissant pour ce type de création. Comme nous ne ferons qu'effleurer les possibilités de Delphi en la matière, n'hésitez pas à vous reporter à l'aide en ligne et aux manuels pour connaître les vastes possibilités offertes par Delphi sur le Web.

Caractéristiques de HTTP et de HTML

Les deux composantes les plus importantes du Web sont HTTP et HTML (*HyperText Markup Language* ou Langage de marquage hypertexte). HTML n'est pas un langage à proprement parler, mais un standard décrivant le format du contenu. Cela signifie que si vous déclarez qu'un document est conforme au standard HTML, des lecteurs peuvent interpréter certaines balises et leur donner un sens. Prenons par exemple un "document" dont le contenu est le suivant :

```
<HTML>
<B> Ceci est en gras </B> <BR>Ceci non.
</HTML>
```

"Ceci non." est dans la police standard. La balise `<B>` active la mise en gras et `</B>` la désactive. La balise `<BR>` indique un saut de ligne. Nous n'entrerons pas ici dans le détail de HTML. Il

existe de nombreux ouvrages consacrés à sa syntaxe ainsi que des programmes permettant de générer du code HTML.

HTTP est un protocole de réseau client/serveur très puissant. Cette puissance vient en partie du fait qu'il permet à un client et à un serveur de communiquer sans qu'il soit nécessaire de maintenir une connexion de réseau persistante. Une URL (`http://www.borland.com` par exemple) est un emplacement universel de ressource qui représente un objet présent sur l'Internet. Si vous utilisez un browser Web pour vous rendre à cette URL, la page Web est transférée vers le browser et l'utilisateur peut alors la visualiser. Cependant, une fois que le chargement de la page s'est achevé, la connexion est rompue. On peut dire, en quelque sorte, que le serveur envoie les données par salves successives. Dans le cas d'une application pour laquelle le réseau est inactif la plupart du temps (comme c'est le cas lorsqu'un utilisateur lit une page Web), le principe de salves de données est tout à fait adapté car il permet au serveur de traiter d'autres requêtes d'informations sans qu'il doive maintenir les ressources correspondant à chacune des connexions inactives.

Le protocole HTTP est orienté transaction : le client effectue une demande de données, après quoi le serveur satisfait à sa requête puis achève la connexion. Le contenu demandé peut être de pratiquement n'importe quel type : documents HTML, images, applications et tout autre objet que le client et le serveur "connaissent" tous deux.

Un autre aspect de la puissance de la technologie Web a trait aux URL, qui sont comme les entrées d'un index universel de l'Internet. Celles-ci permettent d'utiliser en conjonction et de manière intégrée les technologies les plus variées. Une application serveur Web écrite en Delphi peut ainsi être intégrée de manière transparente à une application Web écrite en Perl sur une autre machine utilisant un système d'exploitation différent. Ainsi, par exemple, une application serveur Web Delphi pourrait renvoyer des informations concernant un produit à un browser Web. Lorsque l'utilisateur souhaite acheter le produit, il indique son numéro à une application Perl située sur un serveur UNIX, qui est lié au système d'expédition et de distribution de la société de VPC.

Info

Pour utiliser les exemples de serveur présentés dans cette section, vous aurez besoin d'un serveur Web prenant en charge CGI, ISAPI, NSAPI ou WIN-CGI (ou toute combinaison de ces technologies). Le Microsoft Internet Information Server (IIS) ou le Personal Web Server (fourni avec Windows NT 4.0) gèrent ISAPI et CGI. Il existe d'autres serveurs tournant sur Windows 95, tels que celui fourni avec Front Page 97. Nous examinerons par la suite leurs différences.

Pour utiliser des formulaires actifs, vous aurez besoin d'un browser prenant en charge ActiveX (MSIE 3.0 par exemple).

Le contenu statique de l'Internet

A ses débuts, le Web ne contenait pratiquement que des pages statiques. Autrement dit, lorsqu'un browser Web sélectionnait une URL, le serveur Web renvoyait le document HTML correspondant à cette URL. Le code HTML pouvait également contenir des hyperliens vers d'autres pages Web. Pour l'administrateur système, il suffisait d'enregistrer les fichiers HTML dans une structure de fichier hiérarchisée logiquement. Ce paradigme était parfait pour fournir des informations statiques, mais il ne permettait pas l'interactivité.

Le Listing 14.1 donne un exemple de page Web statique simple.

Listing 14.1 : Une page Web statique simple

```
<HTML>
<TITLE> La cabane du jardinier </TITLE>
<BODY>
<H1> La cabane du jardinier </H1>
<HR>
Avec nous, votre jardin
Les sp&eacute;cialistes mondiaux du rateau
<BR>
<B> Appelez 01-44444444 </B> pour plus d'informations !
<HR>
<A HREF="http://www.jardino.com/tarifs">
Cliquez ici pour voir nos tarifs </A>
</BODY>
</HTML>
```

Lorsque le client (ou le browser) demande `http://www.jardino.com/default.htm`, son serveur Web renvoie le contenu du fichier `default.htm` accompagné d'informations de mise à jour. Ceci peut convenir à La cabane du jardinier, mais le résultat n'est pas spectaculaire et le contenu restera le même à chaque visite.

Créer du contenu dynamique avec Delphi

Imaginons maintenant que l'on souhaite afficher un slogan différent chaque fois que la page est appelée. Un point crucial à garder à l'esprit pour le code HTML généré dynamiquement est que *tout* le traitement s'effectue sur le serveur. Ce dernier ne se contente plus de renvoyer le contenu d'un fichier (comportement statique). Si une requête dynamique est effectuée, il doit traiter un code particulier afin de déterminer ce qui sera envoyé au client. Le serveur peut le

traiter au moyen de différentes techniques. Les deux plus courantes consistent à lancer un exécutable indiquant au serveur ce qu'il convient de renvoyer, ou à appeler une DLL qui exécute un code spécifique et en informe le serveur.

Delphi prend en charge quatre types de processus côté serveur pour la création de HTML dynamique.

Processus	DLL ou EXE
ISAPI	DLL
NSAPI	DLL
CGI	EXE
WIN-CGI	EXE

Lors de cette journée, nous mettrons principalement l'accent sur les ISAPI et les applications CGI. Toutefois, les processus NSAPI sont très similaires aux ISAPI et les programmes WIN-CGI ressemblent aux CGI.

Différences entre ISAPI, NSAPI, CGI et WIN-CGI

Les processus serveur exécutables, tels que les CGI, WIN-CGI, et les DLL en processus, telles que les ISAPI et NSAPI, ont chacun des avantages et des inconvénients spécifiques.

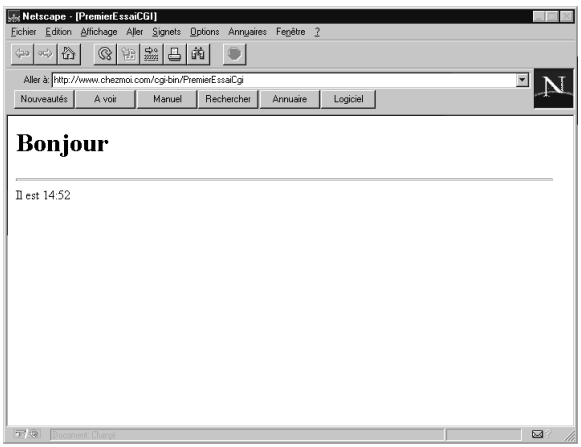
Les applications CGI (*Common Gateway Interface*) constituent le premier type d'application produisant du HTML dynamique. Lorsqu'un serveur Web reçoit une requête de traitement d'un CGI, il transmet à l'application CGI toutes les informations provenant du client, au moyen de variables d'environnement et de l'entrée standard (`stdin`). L'application CGI renvoie le code HTML au client par le biais de la sortie standard (`stdout`). Le processus est en fait plus complexe, puisque des en-têtes et des commandes peuvent être transférés, mais le principe reste identique.

Pour bien voir comment fonctionne une application CGI, nous allons écrire un exécutable de console standard fonctionnant sous forme d'une application CGI simple (ce n'est pas la meilleure manière d'écrire des applications Web en Delphi, mais cet exemple nous permettra de mieux comprendre ce qui se passe). Le code du Listing 14.2 dit bonjour puis donne l'heure. Une fois le programme compilé, il suffit de placer l'exécutable dans un répertoire disposant de privilèges d'exécution sur un serveur Web. L'utilisateur peut alors simplement accéder à l'URL pointant vers l'application (Figure 14.1).

Listing 14.2 : Un exécutable CGI simple utilisant une application de console

```
program consolecgi;

uses
  SysUtils;

begin
   writeln('Content-Type:text/html');
   writeln;
   writeln('<H1> Bonjour </H1> <HR>');
   writeln('Il est '+TimeToStr(Time));
end.
```

Figure 14.1
Un client Web accédant à l'application de console CGI.



Analyse

Cette application se contente d'envoyer les données au client via la sortie standard, au moyen de plusieurs déclarations `Writeln`*. La simplicité du principe peut sembler séduisante. Cependant, avec cette méthode, vous ne pouvez pas tirer parti du cadre de développement Delphi pour les applications serveur. Le cadre de développement (framework) de serveur Web Delphi, dont nous allons parler un peu plus loin, permet d'utiliser une base de code commune aux exécutables et aux processus Web en processus. Il fournit également des routines qui s'acquittent du plus gros du travail lié au développement d'applications CGI et ISAPI/NSAPI.*

L'une des meilleures raisons de préférer CGI à ISAPI (*Internet Server API*) ou NSAPI (*Netscape Server API*) est que la quasi totalité des serveurs Web fonctionnant sous Windows peuvent alors utiliser le même exécutable compilé. Cependant, si les performances sont le critère

principal, CGI n'est pas le plus indiqué. Chaque fois qu'une application client appelle un programme CGI, le serveur Web doit créer un nouveau processus, puis exécuter l'application CGI, renvoyer le résultat au client, et enfin libérer toutes les ressources impliquées. La charge de travail du serveur est importante, et ce particulièrement s'il est très sollicité. Mieux vaut que le serveur Web exécute votre code dans son propre espace de processus sans avoir à lancer un nouvel exécutable chaque fois qu'une requête dynamique est envoyée. C'est précisément le principe qui sous-tend les applications ISAPI (et NSAPI). ISAPI est l'API de serveur du serveur Microsoft et ISAPI est son équivalent pour le serveur de Netscape.

Nouveau

Une application ISAPI est une DLL à threads protégés qui s'exécute dans l'espace de processus du serveur Web. Lorsqu'une requête HTTP appelle la DLL ISAPI, le serveur Web prend un thread dans son pool de thread. Quand elle le lance, il s'exécute dans la DLL. Un pool de thread est un ensemble de threads gérés par le serveur Web, qui peut grandir ou rétrécir dynamiquement en fonction de la charge du serveur. Une fois que le résultat a été envoyé au client, le thread est remis à disposition dans le pool. Cette gestion des ressources est bien plus rationnelle que la création d'un nouveau processus pour chaque exécutable. L'un des inconvénients d'ISAPI est que ce code à threads protégés est difficile à écrire et à tester. De plus, dans le cas d'une application ISAPI, une fois la DLL chargée par le serveur, il est nécessaire d'arrêter ce dernier si vous souhaitez remplacer la DLL.

Le cadre de développement (framework) de serveur Web Delphi

Le dilemme consiste donc à choisir entre les applications CGI et les DLL en processus (ISAPI et NSAPI). Fort heureusement, Delphi vous permet de conserver une base de code commun et de compiler le projet sous forme d'une application CGI, WIN-CGI, ISAPI ou NSAPI.

Nous allons réécrire l'application CGI simple de tout à l'heure, en utilisant cette fois-ci le cadre de développement Web Delphi :

1. Lancez Delphi.
2. Choisissez Fichier, Nouveau dans la boîte de dialogue Nouveaux éléments.
3. Dans l'onglet Nouveau, choisissez Application serveur Web puis cliquez sur OK.
4. On vous demande alors quel type de serveur vous souhaitez construire. Pour l'instant, choisissez `Exécutable autonome CGI`, puis cliquez sur OK.

Vous venez de créer un nouveau projet comportant un module Web et les paramètres permettant de construire l'exécutable CGI. L'étape suivante consiste à ajouter la logique de l'application elle-même.

1. Affectez `CurrentTimeDisp` à la propriété `Name` du `WebModule`.
2. Double-cliquez sur la propriété `Actions` de la fiche. Une boîte de dialogue apparaît alors, du nom de Edition de `CurrentTimeDisp.Actions`. Elle comporte quatre colonnes : Name, PathInfo, Enabled et Default.
3. Cliquez sur le bouton Ajouter. Une nouvelle ligne apparaît alors dans la table des actions.
4. Cliquez sur la nouvelle ligne et utilisez l'Inspecteur d'objet pour renommer le nouvel objet en `DefaultAction`.
5. Double-cliquez sur l'événement `OnAction` dans l'Inspecteur d'objet pour construire le prototype du Gestionnaire d'événements. Le code correspondant s'exécutera lorsque le serveur Web sera appelé.
6. Editez ce Gestionnaire d'événements pour qu'il ait la forme suivante :

```
procedure TCurrentTimeDisp.CurrentTimeDispDefaultActionAction(
  Sender: TObject;
  Request: TWebRequest;
  Response: TWebResponse;
  var Handled: Boolean);
begin
   Response.Content :=  '<H1> Bonjour </H1> <HR>' +
                        'Il est '+
                         TimeToStr(SysUtils.Time);
end;
```

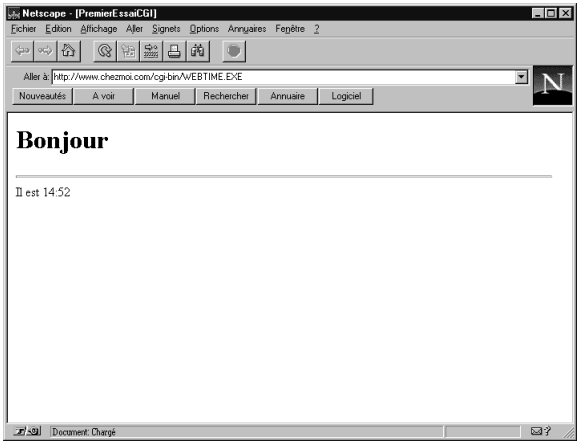
7. Enregistrez et compilez le projet sous le nom `WEBTIME.EXE`.
8. Placez le fichier exécutable dans un "répertoire exécutable" de votre serveur Web. Vous devriez maintenant être en mesure d'appeler cette application CGI comme dans le cas précédent. Il suffit au client de spécifier l'URL correcte, et cette application s'exécutera sur le serveur (voir Figure 14.2.)

Convertir l'application CGI en une DLL ISAPI

La conversion de l'application CGI en une DLL ISAPI (et la conservation d'une base de code commun) est une formalité. Voici la marche à suivre :

1. Fermez tous les projets ouverts.
2. Créez une nouvelle application de serveur Web et choisissez `(ISAPI/NSAPI)` comme type d'application.

Figure 14.2
L'application CGI Hello, construite à l'aide du cadre de développement Web Delphi.



3. Utilisez le Gestionnaire de projet pour supprimer l'unité par défaut et pour ajouter celle créée dans l'exemple précédent.
4. Compilez l'application en une DLL. Et voilà.

Note

N'oubliez pas que si en testant une DLL ISAPI, vous lui apportez une quelconque modification, vous ne pourrez pas la recopier dans le serveur Web sans arrêter puis relancer le service Web. Cela n'est pas nécessaire pour les applications CGI car le nouveau processus est appelé à chaque nouvelle requête. Pour exécuter l'application, il suffit à l'utilisateur de spécifier la DLL dans l'URL. Voici un exemple :

http://www.mysite.com/scripts/webtime.dll

Créer de "vraies" applications Web

Maintenant que nous avons vu que la démarche reste identique pour construire n'importe quel type d'application Web en Delphi, il ne nous reste qu'à examiner la question principale liée au développement d'applications Web, à savoir le choix entre CGI, WIN-CGI, ISAPI ou NSAPI.

Différences entre le modèle de transaction Web et la programmation événementielle

Dans les applications à interface graphique Win32, la programmation est événementielle. Des composants et des formulaires disposent de Gestionnaires d'événements qui répondent à dif-

férentes actions de l'utilisateur. Dans le cas d'une application Web, l'utilisateur ne maintient pas une connexion permanente avec le serveur Web, et chaque transaction doit donc être considérée comme un événement indépendant. Une des difficultés liées à cet état de fait est que l'on ne dispose pas de manière inhérente d'un espace de stockage persistant. Chaque transaction Web n'a pas connaissance de ce qui a pu se passer auparavant.

Prenons l'exemple d'un puzzle écrit en Delphi sous forme d'application Windows. La configuration du jeu peut être placée dans une structure, et à chaque mouvement de l'utilisateur, être mise à jour. Ceci est plus difficile à réaliser dans une application serveur Web car il est alors nécessaire de "simuler" un stockage permanent. Dans l'exemple d'application présenté dans la partie suivante, on simulera ce stockage persistant en envoyant toutes les informations nécessaires au serveur chaque fois que le client effectue une nouvelle requête.

Exemple : un puzzle

Voici un puzzle que vous connaissez peut-être déjà. Il s'agit d'une grille trois par trois contenant huit pièces. L'un des emplacements est vide. Les pièces peuvent coulisser de telle manière que toute pièce située à côté de l'emplacement vide peut glisser pour l'occuper. Le but du jeu est de les disposer dans l'ordre, avec l'emplacement vide situé dans le coin inférieur droit, comme dans l'exemple ci-après :

```
3  2  6        3  2  6                      1  2  3
1  8  4  →     1  8  ×  Et finalement →     4  5  6
7  5  ×        7  5  4                      7  8  ×
```

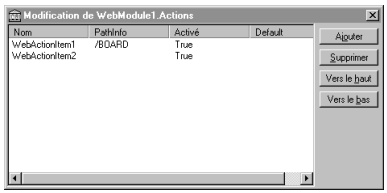
Cet exemple illustre trois concepts très importants pour le développement d'applications serveur Web en Delphi :

- Comment permettre à une unique application d'effectuer plusieurs tâches, en utilisant des informations de chemin d'accès pour déclencher plusieurs événements dans la propriété `Action` d'un formulaire Web.
- Comment lire les informations passées du client au serveur, à l'aide de l'objet `TWebRequest`.
- Comment simuler le stockage permanent en passant des informations provenant du client.

Pour construire cette application, suivez ces étapes :

1. Créez une nouvelle application serveur Web.
2. Dans la propriété `Actions` du formulaire Web, définissez deux objets `Action`, et affectez `\BOARD` à la propriété `Path` de l'un d'entre eux (celle de l'autre reste vide), comme le montre la Figure 14.3.

Figure 14.3
L'éditeur Action *pour le module Web montrant les deux objets* Action.



Dans le Listing 14.3 figure le code source correspondant au reste de l'implémentation. Les fonctions `CanMove`, `BuildRef` et `DrawBoard` sont des utilitaires indépendantes que vous pouvez entrer manuellement dans l'unité. Vous pouvez ajouter les deux événements `action` en double-cliquant sur l'élément d'événement `OnAction` de ces deux actions. On suppose ici que l'exécutable résidera dans le répertoire virtuel `scripts` et qu'il aura pour nom `TileGame.EXE`.

Listing 14.3 : Un jeu de taquin Web

```
unit tgUnit;

interface

uses
  Windows, Messages, SysUtils, Classes, HTTPApp;

type
  TWebModule1 = class(TWebModule)
    procedure WebModule1BoardAction(Sender: TObject;
      Request: TWebRequest;
      Response: TWebResponse; var Handled: Boolean);
    procedure WebModule1WebActionItem1Action(Sender: TObject;
      Request: TWebRequest; Response: TWebResponse;
      var Handled: Boolean);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  WebModule1: TWebModule1;

implementation
```

```
{$R *.DFM}
function CanMove(X,Y:integer):boolean;
Var
   PossibleMoves : Array[0..8,0..8] of boolean;
   CountX,CountY : integer;
{En supposant que X est la pièce manquante, cette fonction répond à la
question : }
{La pièce Y peut-elle aller dans l'espace vide X ?}
begin
  {on marque tous les mouvements comme impossibles}
  For CountX := 0 to 8 do
    For CountY := 0 to 8 do
      PossibleMoves[CountX,CountY] := False;
  {On permet aux pièces adjacentes de glisser}
  {Pièce 0 manquante} PossibleMoves[0,1] := True;
                      PossibleMoves[0,3] := True;
  {Pièce 1 Manquante} PossibleMoves[1,0] := True;
                      PossibleMoves[1,2] := True;
                      PossibleMoves[1,4] := True;
  {Pièce 2 Manquante} PossibleMoves[2,1] := True;
                      PossibleMoves[2,5] := True;
  {Pièce 3 manquante} PossibleMoves[3,0] := True;
                      PossibleMoves[3,4] := True;
                      PossibleMoves[3,6] := True;
  {Pièce 4 manquante} PossibleMoves[4,1] := True;
                      PossibleMoves[4,3] := True;
                      PossibleMoves[4,5] := True;
                      PossibleMoves[4,7] := True;
  {Pièce 5 manquante} PossibleMoves[5,2] := True;
                      PossibleMoves[5,4] := True;
                      PossibleMoves[5,8] := True;
  {Pièce 6 manquante} PossibleMoves[6,3] := True;
                      PossibleMoves[6,7] := True;
  {Pièce 7 manquante} PossibleMoves[7,4] := True;
                      PossibleMoves[7,6] := True;
                      PossibleMoves[7,8] := True;
  {Pièce 8 manquante} PossibleMoves[8,5] := True;
                      PossibleMoves[8,7] := True;
  CanMove := PossibleMoves[X,Y];
end;
```

```
function BuildRef(Tiles : Array of integer;Tile1,Tile2:integer):string;
{Cette procédure construit la référence pour la pièce sélectionnée }
{Elle échange la pièce sur laquelle l'utilisateur a cliqué avec l'espace
vide }
Var
  Count    : integer;
  RefBuilt : string;

begin
  {La ligne suivante doit faire référence à une DLL si l'application est
ISAPI}
  RefBuilt:= '/scripts/TILEGAME.EXE/BOARD?';
  for Count := 0 to 8 do
  begin
     if Tiles[Count]=Tile1 then
      RefBuilt := RefBuilt + IntToStr(Tile2)
     else if Tiles[Count]=Tile2 then
      RefBuilt := RefBuilt + IntToStr(Tile1)
     else
      RefBuilt := RefBuilt + IntToStr(Tiles[Count]);
  end;
  BuildRef := RefBuilt;
end;

procedure DrawBoard(Tiles : Array of integer;Response: TWebResponse);
{ Ici on dessine le plateau de jeu. On a passé un objet TWebResponse}
{et on a affecté à sa propriété content le code HTML correspondant }
Var
    BlankSpace : Integer;
    Count      : Integer;

begin
 { On place le code HTML de titre}
 Response.content:='<TITLE> Jeu de taquin </TITLE>';
 Response.content:=Response.content+'<H1> Jeu de taquin </H1><HR>';
 Response.content:=Response.content+'<CENTER><TABLE BORDER = 1><TR>';
 BlankSpace := 9;

 {On regarde quelle est la pièce manquante (pièce 0)}
 For Count := 0 to 8 do
  if Tiles[Count] = 0 then BlankSpace := Count;
```

```
 {on parcourt les 9 emplacements }
 For Count := 0 to 8 do
 begin
   {on regarde si la pièce courante est la manquante }
   if Tiles[Count] = 0 then
     Response.content := Response.content +'<TD>X<TD>'
   {On regarde si on peut faire glisser la pièce courante dans l'emplacement
vide }
   else if CanMove(BlankSpace,Count) then
           begin
             Response.content := Response.content +'<TD> <A HREF="'+
                                 BuildRef(Tiles,0,Tiles[Count])+'">'+
                                 IntToStr(Tiles[Count])+' </A><TD>'
           end
         else
           Response.content := Response.content +'<TD>'+
                               IntToStr(Tiles[Count])+'<TD>';
   { En fin de chaque colonne, on termine la colonne et on en commence une
nouvelle }
   if Count Mod 3 = 2 then
     Response.content := Response.content + '</TR><TR>';
 end;
 {on referme le tableau }
 Response.content := Response.content +
                     '</TR></TABLE></CENTER><HR> Tilegame V.1.0';
end;

procedure TWebModule1.WebModule1BoardAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
{Lorsque l'utilisateur effectue un mouvement, le chemin Bord est invoqué}
{Cette procédure tire la nouvelle configuration du plateau de la chaîne de
requête,}
{puis regarde si l'utilisateur a gagné et s'il n'est donc pas nécessaire de
redessiner le plateau }
var
   Tiles : array[0..8] of integer;
   Count : integer;
   TempTile : string;

begin
```

```
     {Gagné ? }
     if Request.Query = '123456780' THEN
      begin
       Response.Content := '<H1> Vous avez gagn&eacute; ! </H1> <HR>';
       Response.Content := Response.Content+
                           '<A HREF="/scripts/tilegame.exe">'+
                           'Cliquez ici pour rejouer </A> <HR>';
      end
     else
      {Sinon, on prend la configuration dans Request.Query et on redessine
le plateau }
      begin
       for Count := 0 to 8 do
       begin
         TempTile :=  copy(Request.Query,Count+1,1);
         Tiles[Count] := StrToInt(TempTile);
       end;
       DrawBoard(Tiles,Response);
      end;
end;

procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
{Cette procédure est appelée lorsqu'un nouveau jeu est demandé (chemin vide) }
var
   Count      : integer;
   BlankCount : integer;
   BlankTile  : integer;
   TryMove    : integer;
   Tiles      : Array[0..8] of Integer;

begin
 Randomize;
 { On initialise le plateau }
 For Count := 0 to 7 do
   Tiles[Count] := Count +1;
 { La dernière pièce est manquante (représentée par 0)}
 Tiles[8] := 0;

 {On mélange le plateau en faisant glisser aléatoirement les pièces 100 fois
}
```

```
 For Count := 0 to 100 do
 begin
   {On cherche la pièce manquante }
   BlankTile := 0;
   for BlankCount := 0 to 8 do
     if Tiles[BlankCount] = 0 then
       BlankTile := BlankCount;
   repeat
     TryMove := Random(9);
   until CanMove(BlankTile,TryMove);
   Tiles[BlankTile] := Tiles[TryMove];
   Tiles[TryMove]   := 0;
 end;

 {On dessine le plateau de départ }
 DrawBoard(Tiles,Response);
end;

end.
```

Analyse

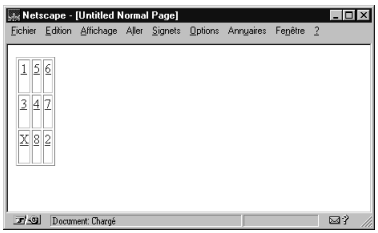
Le module Web peut prendre en charge plusieurs chemins d'accès. Un chemin est la portion de l'URL de requête HTTP qui suit l'application mais précède la chaîne de requête. Ainsi, l'URL HTTP suivant :

```
http://www.canino.com/chiens.exe/RECH_RACE?TYPECHIEN=Cocker
```

contiendra le chemin `Path=/"RECH_RACE"` *et une chaîne de requête* `TYPECHIEN=Cocker`*. En passant plusieurs chemins à une application Web, vous pouvez donc permettre à celle-ci d'implémenter plusieurs fonctions. Nous utiliserons deux chemins dans notre application. Le premier est vide, ce qui signifie que si l'application est invoquée sans spécifier de chemin, le Gestionnaire d'événements* `OnAction` *sera appelé. Dans le jeu de taquin, celui-ci mélange aléatoirement le puzzle et débute une nouvelle partie. On utilise la fonction* `BuildRed` *pour placer des ancres sous toutes les pièces qui peuvent être déplacées sur l'emplacement vide (la logique correspondante se trouve dans la fonction* `CanMove`*). L'ancre contient un chemin* `/BOARD` *et une chaîne de requête égale à ce à quoi ressemblerait le puzzle si cette pièce était choisie. Une ancre HTML est l'ensemble de balises qui fait d'elle un lien. En substance, on regarde un coup en avance, et ce pour tous les coups possibles à un instant donné. Lorsqu'une requête de chemin* `\BOARD` *est entrée, l'application appelle le Gestionnaire d'événements* `OnAction` *associé à ce chemin. Le Gestionnaire d'événements de chemin* `\BOARD` *regarde si l'utilisateur a gagné. Si c'est le cas, un message est affiché. Sinon, on redessine le puzzle dans la configuration transmise par la chaîne de requête.*

La Figure 14.4 donne un exemple d'exécution de cette application.

Figure 14.4
Le jeu de taquin sur le Web.



La chaîne de requête `Query` provient de l'objet `TWebRequest`, qui est passé au Gestionnaire d'événements `PathInfo`. Cet objet contient toutes les informations que le client (un browser Web, la plupart du temps) a envoyé au serveur. Le rôle de l'application est de prendre dedans toutes les informations nécessaires, d'exécuter la logique de l'objet, puis de renvoyer des informations au client via l'objet `TWebResponse`.

Obtenir des informations du client à l'aide de formulaires

Nous avons vu des applications serveur qui s'exécutent sans que l'utilisateur n'ait effectué aucune entrée, comme l'application Hello, et d'autres qui examinaient la chaîne de requête pour y trouver des entrées personnalisées (le jeu de taquin). Dans cette partie, nous verrons comment une application peut utiliser des formulaires pour recevoir et traiter des données issues de browsers Web.

Les formulaires utilisent des composants de saisie standard tels que les boutons, les boîtes de liste déroulantes, les cases à cocher, et autres pour obtenir des données de l'utilisateur, et pour les envoyer au serveur Web. La Figure 14.5 donne un exemple de page simple contenant un formulaire de saisie de numéro de téléphone.

Le client prend toutes les entrées saisies dans le formulaire, les code sous un format particulier, puis les envoie au serveur. Le Listing 14.4 montre le code source correspondant à la page HTML de la Figure 14.5.

Listing 14.4 : Code HTML pour un formulaire simple

```
<html>
<head>
<meta http-equiv="Content-Type"
content="text/html; charset=iso-8859-1">
<title>Jardino</title>
```

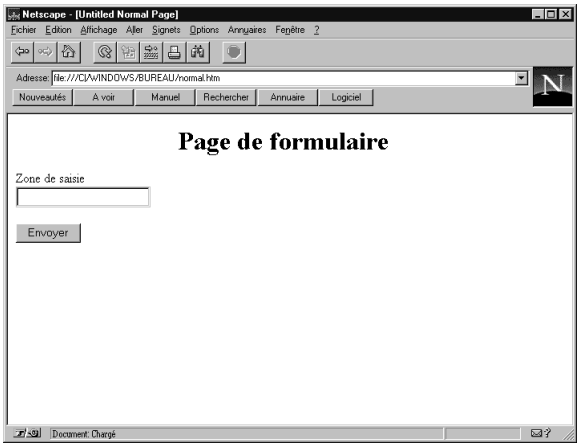


Figure 14.5
Une page Web comportant un formulaire simple.

```
</head>

<body bgcolor="#FFFFFF">

<h2>Tout pour le jardinage</h2>

<hr>

<h4>Pour recevoir de plus amples informations sur nos services, veuillez
entrer votre nom et votre num&eacute;ro de t&eacute;l&eacute;phone :</h4>

<form action="http://www.jardino.com/scripts/maillist.exe"
method="GET" name="Mailing List">
    <div align="center"><center><table border="0">
        <tr>
            <td>Nom : </td>
            <td><input type="text" size="20" name="Name"></td>
        </tr>
        <tr>
            <td>T&eacute;l&eacute;phone :</td>
            <td><input type="text" size="20" name="Phone"
            value="XX-XX-XX-XX-XX"></td>
        </tr>
        <tr>
            <td align="center" colspan="2"><input type="submit"
```

```
            name="B1" value="Envoyer"></td>
        </tr>
    </table>
    </center></div>
</form>

<p> </p>
</body>
</html>¤
```

Analyse

Ce code contient trois balises de saisie, qui correspondent aux deux boîtes de saisie et au bouton poussoir du formulaire. Vous pouvez vous représenter les composants de saisie comme des variables qui se verront affecter des valeurs lorsque le formulaire sera envoyé. Dans notre exemple, `Name` *et* `Phone` *seront envoyées au serveur, et contiendront les informations saisies. Ces dernières sont transmises à l'application du serveur spécifiée dans la section* `action`*, c'est-à-dire ici à l'URL suivante :*

```
http://www.jardino.com/scripts/maillist.exe
```

L'une des parties importantes du formulaire est la section `method` de la balise `form`. Cette balise spécifie que les données du formulaire doivent être, soit incluses dans l'URL de l'application sous forme d'une chaîne de requête, soit être passées indépendamment de la chaîne de requête. La méthode `Get` envoie les données dans l'URL, tandis que la méthode `Post` les envoie sous forme de contenu (via l'entrée standard d'une application CGI). La méthode utilisée détermine la valeur de certaines propriétés de l'objet `TWebRequest`.

Pour savoir quelles sont les données réellement passées à l'application Web, il suffit de créer une application Web simple affichant la chaîne de requête envoyée par un formulaire. Cette application n'a besoin que d'un Gestionnaire de chemin par défaut contenant le Gestionnaire d'événements `OnAction` suivant :

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := Request.Query;
end;
```

Compilez ce programme sous le nom `MAILLIST.EXE` et placez-le sur le serveur Web dans le répertoire scripts (ou dans tout autre répertoire exécutable). Que se passe-t-il si vous entrez `Albert d'Artagnan` comme nom et `01-44-44-44-44` comme numéro de téléphone dans le formulaire généré par le Listing 14.4 ? Le serveur Web traite la requête et affiche la chaîne suivante :

```
Name=Albert+d%92Artagnan&Phone=01-44-44-44-44&B1=Submit
```

On retrouve vaguement le nom et le numéro de téléphone, mais également des caractères étranges. C'est la chaîne codée envoyée par le formulaire. Le mécanisme de codage est le suivant :

- Tous les champs (ou variables) sont séparés par des `&`.
- Les noms de champs sont séparés des données associées par un signe égal (`=`).
- Les espaces sont convertis en signes plus (`+`).
- Les caractères non alphanumériques sont convertis en un signe de pourcentage suivi par la valeur ASCII du caractère en hexadécimal (`%92` par exemple).

Pour décoder une chaîne, il suffit de suivre la procédure ci-après. On commence par décomposer la chaîne selon les `&` :

```
VARIABLE 1: Name=Albert+d%92Artagnan
VARIABLE 2: Phone=01-44-44-44-44
VARIABLE 3: B1=Submit
```

Ensuite, on décompose chaque variable selon le signe `=` :

```
VARIABLE(Name) : Albert+d%92Artagnan
VARIABLE(Phone): 01-44-44-44-44
VARIABLE(B1):    Submit
```

On transforme les `+` en espaces :

```
VARIABLE(Name) : Albert d%92Artagnan
VARIABLE(Phone): 01-44-44-44-44
VARIABLE(B1):    Submit
```

Et on convertit les `%xx` pour obtenir les caractères correspondants :

```
VARIABLE(Name) : Albert d'Artagnan
VARIABLE(Phone): 01-44-44-44-44
VARIABLE(B1):    Submit
```

En fait, vous n'avez pas besoin d'écrire du code qui se charge d'extraire les données stockées dans la chaîne de requête.

Modifions le Gestionnaire d'événements `OnAction` afin de voir comment utiliser des méthodes pour décoder les chaînes :

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := '<HTML>Var1:'+Request.QueryFields.Strings[0];
  Response.Content := Response.Content +'<br>Var2:'+
                      Request.QueryFields.Strings[1];
```

```
  Response.Content := Response.Content +'<br>Var3:'+
                      Request.QueryFields.Strings[2];
  Response.Content := Response.Content +'</HTML>';
end;
```

Le résultat renvoyé par l'application Web est alors le suivant :

```
Var1:Name=Albert d'Artagnan
Var2:Phone=01-44-44-44-44
Var3:B1=Submit
```

Delphi s'est chargé de tout le travail en décodant la chaîne de requête. En effet, on a utilisé ici les `Tstrings QueryFields`. Pour accéder à la chaîne brute, vous devez utiliser la propriété `Request.Query`, et pour les champs décodés, la propriété `Request.QueryFields`. Si le formulaire avait été envoyé avec une méthode `post` plutôt que `get`, l'application devrait utiliser les propriétés `Request.Content` et `Request.ContentFields`.

Pour achever notre application, ajoutez un code qui place le nom et le numéro de téléphone du client dans un fichier puis envoie un message de confirmation. Le code définitif de l'application se trouve Listing 14.5.

Listing 14.5 : Application enregistrant le nom et le numéro de téléphone du client

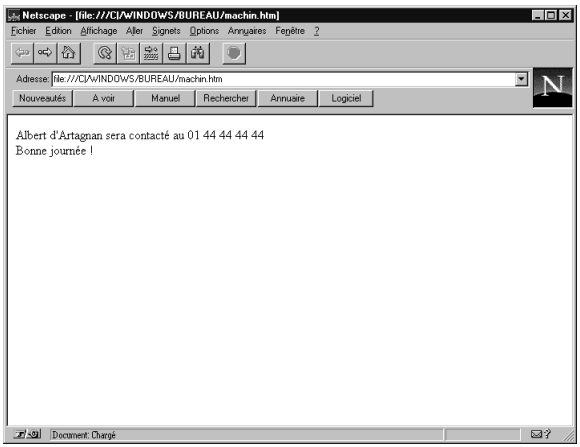
```
unit unitMailList;

interface

uses
  Windows, Messages, SysUtils, Classes, HTTPApp;

type
  TWebModule1 = class(TWebModule)
    procedure WebModule1WebActionItem1Action(Sender: TObject;
      Request: TWebRequest; Response: TWebResponse;
      var Handled: Boolean);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  WebModule1: TWebModule1;
```

```
implementation

{$R *.DFM}

procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);

var
  F        : TextFile;
  Callline : string;

begin
  AssignFile(F,'calllist.txt');
  if FileExists('calllist.txt') then
    Append(F)
   else
    ReWrite(F);
  callline := copy(Request.QueryFields.Strings[0],6,
                   length(Request.QueryFields.Strings[0])-5);
  callline := callline +' sera contacté à '+copy(
              Request.QueryFields.Strings[1],7,
              length(Request.QueryFields.Strings[1])-6);
  Response.Content := '<HTML>'+callline+'<br> Bonne journée !';
  Response.Content := Response.Content +'</HTML>';
  writeln(F,callline);
  Close(F);
end;

end.
```

Analyse

Lorsque le client Web envoie le formulaire, la procédure `WebModule1WebActionItem1Action` est appelée. Elle décode ses entrées et les inscrit dans un fichier. L'appel à `FileExists` détermine si le fichier existe déjà. Si c'est le cas, on ajoute les informations à la fin du fichier en appelant `Append(F)`. Sinon, on crée un nouveau fichier, en appelant la procédure `ReWrite(F)`. Comme les noms de fichiers ont une longueur constante, vous pouvez extraire les données de la propriété `QueryString` en utilisant la fonction `copy` et en spécifiant une valeur fixe comme deuxième paramètre. La Figure 14.6 montre la réponse de l'application Web après envoi du formulaire.

Figure 14.6
Le message de confirmation de l'application.



Utiliser des formulaires actifs au niveau du client

La plupart des browsers Web permettent maintenant au client d'exécuter du code plutôt que de laisser la machine serveur assumer toute la charge du traitement. Les deux principales technologies qui rendent ceci possible sont Java et ActiveX. Il est préférable de laisser le traitement au serveur si vous souhaitez atteindre la plus large audience possible. Nombreux sont les browsers qui en sont incapables, ou pour lesquels cette fonctionnalité a été désactivée. Le traitement client est conseillé s'il existe une interaction constante entre l'utilisateur et l'application, ou si vous disposez déjà d'une application Windows que vous souhaitez convertir en un outil Web. Delphi vous permet de créer des composants ActiveX utilisables dans des scripts de formulaires Web et d'utiliser des formulaires actifs. Cette dernière technique vous permet de créer un formulaire sous forme de composant ActiveX. Vous pouvez alors l'incorporer dans une page Web utilisant ActiveX. Créons ensemble un formulaire actif, puis une page Web dans laquelle sera incorporé ce formulaire.

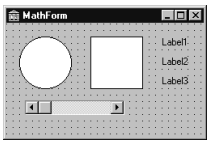
1. Fermez tous les projets déjà ouverts, puis sélectionnez Fichier, Nouveau.
2. Dans la boîte de dialogue Nouveaux éléments, sélectionnez l'onglet ActiveX, puis choisissez `Bibliothèque ActiveX`.
3. Choisissez encore Fichier, Nouveau, sélectionnez l'onglet ActiveX, puis `ActiveForm`. L'Expert ActiveForm apparaît alors et vous demandera d'entrer un nom pour le nouvel ActiveX ainsi qu'un autre pour l'unité d'implémentation. Dans notre exemple, le nom de

l'ActiveX est `MathForm` et celui du fichier `MathFormImpl.pas`. Vous n'avez pas besoin de cocher les autres options de la boîte.

4. Une nouvelle application semble avoir été créée, mais la fiche qui apparaît à l'écran correspond en fait à un composant ActiveX. Vous pouvez à présent placer des composants et ajouter des codes sur cette fiche, comme si vous écriviez une application Windows.

Dans notre exemple, imaginons qu'un professeur de maths utilise le Web pour envoyer ses leçons et proposer quelques informations supplémentaires. La leçon du jour a trait aux aires des cercles et des carrés. Il serait du plus bel effet que la page Web contienne un cercle et un carré, ainsi qu'une barre de curseur permettant d'ajuster leurs aires de manière à ce qu'elles soient toujours égales (voir Figure 14.7). Ceci est très difficile à réaliser à l'aide d'une application de serveur Web, mais ne pose aucun problème si on utilise des fiches ActiveForm.

Figure 14.7
La page Web de Leçon de math, utilisant un ActiveForm.



Poursuivons la création de l'application :

1. Placez six composants sur la fiche : deux composants de forme, trois composants label et un scrollbar.
2. Nommez un des composants de forme `Circle` et l'autre `Square`.
3. Affectez `stCircle` à la propriété `shape` de l'objet `Circle`, et `stRectangle` à celle de l'objet `Square`. Affectez `Area` à la propriété `name` de `ScrollBar` et `100` et `500` respectivement aux propriétés `min` et `max`.
4. Implémentez l'événement `OnChange` de la barre de curseur et l'événement `OnCreate` de la fiche, en suivant le Listing 14.6. Ce sont les seules procédures à ajouter. Delphi génère automatiquement le reste du code.

Listing 14.6 : L'application Math utilisant des ActiveForms

```
unit MathFormImpl;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs,
  ActiveX, AxCtrls, MathLib_TLB, ExtCtrls, StdCtrls;

```

```
type
  TMathForm = class(TActiveForm, IMathForm)
    Area: TScrollBar;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Square: TShape;
    Circle: TShape;
    procedure FormCreate(Sender: TObject);
    procedure AreaChange(Sender: TObject);
  private
    { Déclarations privées }
    FEvents: IMathFormEvents;
    procedure ActivateEvent(Sender: TObject);
    procedure ClickEvent(Sender: TObject);
    procedure CreateEvent(Sender: TObject);
    procedure DblClickEvent(Sender: TObject);
    procedure DeactivateEvent(Sender: TObject);
    procedure DestroyEvent(Sender: TObject);
    procedure KeyPressEvent(Sender: TObject; var Key: Char);
    procedure PaintEvent(Sender: TObject);
  protected
    { Déclarations protégées }
    procedure EventSinkChanged(const EventSink: IUnknown); override;
    procedure Initialize; override;
    function CloseQuery: WordBool; safecall;
    function Get_Active: WordBool; safecall;
    function Get_AutoScroll: WordBool; safecall;
    function Get_AxBorderStyle: TxActiveFormBorderStyle; safecall;
    function Get_Caption: WideString; safecall;
    function Get_Color: TColor; safecall;
    function Get_Cursor: Smallint; safecall;
    function Get_DropTarget: WordBool; safecall;
    function Get_Enabled: WordBool; safecall;
    function Get_Font: Font; safecall;
    function Get_HelpFile: WideString; safecall;
    function Get_KeyPreview: WordBool; safecall;
    function Get_ModalResult: Integer; safecall;
    function Get_PixelsPerInch: Integer; safecall;
    function Get_PrintScale: TxPrintScale; safecall;
    function Get_Scaled: WordBool; safecall;
```

```
    function Get_Visible: WordBool; safecall;
    function Get_WindowState: TxWindowState; safecall;
    function ShowModal: Integer; safecall;
    procedure Close; safecall;
    procedure DisableAutoRange; safecall;
    procedure EnableAutoRange; safecall;
    procedure Print; safecall;
    procedure Set_AutoScroll(Value: WordBool); safecall;
    procedure Set_AxBorderStyle(Value: TxActiveFormBorderStyle);
      safecall;
    procedure Set_Caption(const Value: WideString); safecall;
    procedure Set_Color(Value: TColor); safecall;
    procedure Set_Cursor(Value: Smallint); safecall;
    procedure Set_DropTarget(Value: WordBool); safecall;
    procedure Set_Enabled(Value: WordBool); safecall;
    procedure Set_Font(const Value: Font); safecall;
    procedure Set_HelpFile(const Value: WideString); safecall;
    procedure Set_KeyPreview(Value: WordBool); safecall;
    procedure Set_ModalResult(Value: Integer); safecall;
    procedure Set_PixelsPerInch(Value: Integer); safecall;
    procedure Set_PrintScale(Value: TxPrintScale); safecall;
    procedure Set_Scaled(Value: WordBool); safecall;
    procedure Set_Visible(Value: WordBool); safecall;
    procedure Set_WindowState(Value: TxWindowState); safecall;
  public
    { Déclarations publiques }
  end;

implementation

uses ComServ;

{$R *.DFM}

{ TMathForm }

procedure TMathForm.EventSinkChanged(const EventSink: IUnknown);
begin
  FEvents := EventSink as IMathFormEvents;
end;
```

```
procedure TMathForm.Initialize;
begin
  OnActivate := ActivateEvent;
  OnClick := ClickEvent;
  OnCreate := CreateEvent;
  OnDblClick := DblClickEvent;
  OnDeactivate := DeactivateEvent;
  OnDestroy := DestroyEvent;
  OnKeyPress := KeyPressEvent;
  OnPaint := PaintEvent;
end;

function TMathForm.CloseQuery: WordBool;
begin

end;

function TMathForm.Get_Active: WordBool;
begin
  Result := Active;
end;

function TMathForm.Get_AutoScroll: WordBool;
begin
  Result := AutoScroll;
end;

function TMathForm.Get_AxBorderStyle: TxActiveFormBorderStyle;
begin
  Result := Ord(AxBorderStyle);
end;

function TMathForm.Get_Caption: WideString;
begin
  Result := WideString(Caption);
end;

function TMathForm.Get_Color: TColor;
begin
  Result := Color;
end;
```

```
function TMathForm.Get_Cursor: Smallint;
begin
  Result := Smallint(Cursor);
end;

function TMathForm.Get_DropTarget: WordBool;
begin
  Result := DropTarget;
end;

function TMathForm.Get_Enabled: WordBool;
begin
  Result := Enabled;
end;

function TMathForm.Get_Font: Font;
begin
  GetOleFont(Font, Result);
end;

function TMathForm.Get_HelpFile: WideString;
begin
  Result := WideString(HelpFile);
end;

function TMathForm.Get_KeyPreview: WordBool;
begin
  Result := KeyPreview;
end;

function TMathForm.Get_ModalResult: Integer;
begin
  Result := Integer(ModalResult);
end;

function TMathForm.Get_PixelsPerInch: Integer;
begin
  Result := PixelsPerInch;
end;
```

```
function TMathForm.Get_PrintScale: TxPrintScale;
begin
  Result := Ord(PrintScale);
end;

function TMathForm.Get_Scaled: WordBool;
begin
  Result := Scaled;
end;

function TMathForm.Get_Visible: WordBool;
begin
  Result := Visible;
end;

function TMathForm.Get_WindowState: TxWindowState;
begin
  Result := Ord(WindowState);
end;

function TMathForm.ShowModal: Integer;
begin

end;

procedure TMathForm.Close;
begin

end;

procedure TMathForm.DisableAutoRange;
begin

end;

procedure TMathForm.EnableAutoRange;
begin

end;

procedure TMathForm.Print;
```

```
begin

end;

procedure TMathForm.Set_AutoScroll(Value: WordBool);
begin
  AutoScroll := Value;
end;

procedure TMathForm.Set_AxBorderStyle(Value: TxActiveFormBorderStyle);
begin
  AxBorderStyle := TActiveFormBorderStyle(Value);
end;

procedure TMathForm.Set_Caption(const Value: WideString);
begin
  Caption := TCaption(Value);
end;

procedure TMathForm.Set_Color(Value: TColor);
begin
  Color := Value;
end;

procedure TMathForm.Set_Cursor(Value: Smallint);
begin
  Cursor := TCursor(Value);
end;

procedure TMathForm.Set_DropTarget(Value: WordBool);
begin
  DropTarget := Value;
end;

procedure TMathForm.Set_Enabled(Value: WordBool);
begin
  Enabled := Value;
end;

procedure TMathForm.Set_Font(const Value: Font);
begin
```

```
  SetOleFont(Font, Value);
end;

procedure TMathForm.Set_HelpFile(const Value: WideString);
begin
  HelpFile := String(Value);
end;

procedure TMathForm.Set_KeyPreview(Value: WordBool);
begin
  KeyPreview := Value;
end;

procedure TMathForm.Set_ModalResult(Value: Integer);
begin
  ModalResult := TModalResult(Value);
end;

procedure TMathForm.Set_PixelsPerInch(Value: Integer);
begin
  PixelsPerInch := Value;
end;

procedure TMathForm.Set_PrintScale(Value: TxPrintScale);
begin
  PrintScale := TPrintScale(Value);
end;

procedure TMathForm.Set_Scaled(Value: WordBool);
begin
  Scaled := Value;
end;

procedure TMathForm.Set_Visible(Value: WordBool);
begin
  Visible := Value;
end;

procedure TMathForm.Set_WindowState(Value: TxWindowState);
begin
  WindowState := TWindowState(Value);
```

```
end;

procedure TMathForm.ActivateEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnActivate;
end;

procedure TMathForm.ClickEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnClick;
end;

procedure TMathForm.CreateEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnCreate;
end;

procedure TMathForm.DblClickEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnDblClick;
end;

procedure TMathForm.DeactivateEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnDeactivate;
end;

procedure TMathForm.DestroyEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnDestroy;
end;

procedure TMathForm.KeyPressEvent(Sender: TObject; var Key: Char);
var
  TempKey: Smallint;
begin
  TempKey := Smallint(Key);
  if FEvents <> nil then FEvents.OnKeyPress(TempKey);
  Key := Char(TempKey);
end;
```

```
procedure TMathForm.PaintEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnPaint;
end;


procedure TMathForm.FormCreate(Sender: TObject);
begin
   Area.Position := 250;
end;

procedure TMathForm.AreaChange(Sender: TObject);
begin
     Circle.Width  :=trunc(2.0 *(sqrt(Area.Position/3.14159)));
     Circle.Height :=trunc(2.0 *(sqrt(Area.Position/3.14159)));
     Square.Width  :=trunc(sqrt(Area.Position));
     Square.Height :=trunc(sqrt(Area.Position));
     Label1.Caption :=  'Aire :'+
                        IntToStr(Area.Position);
     Label2.Caption :=  'Rayon du cercle :'+
                        FloatToStr(2.0 *
                                  (sqrt(Area.Position/3.14159))/2);
     Label3.Caption := 'Hauteur et longueur du carré :'+
                        FloatToStr(sqrt(Area.Position));
end;

initialization
  TActiveFormFactory.Create(
    ComServer,
    TActiveFormControl,
    TMathForm,
    Class_MathForm,
    1,
    '',
    OLEMISC_SIMPLEFRAME or OLEMISC_ACTSLIKELABEL);
end.
```

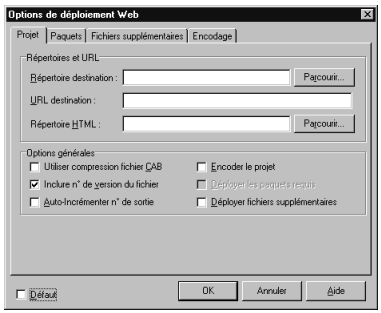
Analyse

Cette application sera compilée sous forme d'OCX ActiveX. On l'utilisera comme base de code pour la fiche, sous forme de composant ActiveX. Lorsqu'une application Web référence l'ID de classe (`CLSID`) du composant que vous avez créé, l'ActiveForm sera incorporée dans la page Web du client. Il est important de noter que les composants ActiveX peuvent égale-

ment être inserés dans des applications non liées au Web. Une fois que le client a incorporé le composant, ce dernier est exécuté comme toute autre fiche Delphi le serait dans une application. Lorsqu'il fait glisser le curseur, l'aire correspondante du carré et du cercle est calculée, et les dimensions des composants de forme sont modifiées en conséquence. L'application affiche également l'aire et les dimensions des formes à l'aide des composants label.

Delphi propose une fonctionnalité très puissante, du nom de "Déploiement Web ". Celle-ci vous permet de construire rapidement une page Web pour tester le composant, et pour placer l'OCX ActiveX sur votre serveur Web. Avant d'utiliser cette option, vous devez configurer les options de Déploiement Web en sélectionnant Projet, Options de déploiement Web dans le menu principal. La Figure 14.8 indique les options adaptées à notre exemple. En plus du déploiement de l'application, cette fonction permet également de signer numériquement l'ActiveForm. Pour plus de détails sur l'*authenticode* de Microsoft, consultez les manuels en ligne de Delphi.

Figure 14.8
La boîte de dialogue Options de déploiement Web.



Les trois champs de la boîte de dialogue Options de déploiement Web qu'il est nécessaire de remplir sont Répertoire destination, URL Destination et Répertoire HTML. Le répertoire cible est celui dans lequel l'OCX ActiveX sera placé. Il correspond au répertoire d'URL qui figure dans le champ URL Destination. Ainsi, si sur votre serveur le répertoire `D:\InetPub\WWWROOT` correspond à `http://www.jardino.com`, vous spécifieriez `D:\InetPub\WWWROOT` comme répertoire cible et `http://www.jardino.com` comme URL cible. Remarquez au passage que, dans cet exemple, le répertoire racine virtuelle du serveur Web se trouve à l'emplacement physique `D:\InetPub\WWWROOT`. Le champ Répertoire HTML spécifie l'emplacement d'un exemple de document incorporant l'ActiveForm. Vous pouvez utiliser cet exemple de document comme point de départ pour votre page utilisant l'ActiveForm. Pour que Delphi place les fichiers sur le serveur Web, sélectionnez Projet, Déploiement Web.

Le Listing 14.7 correspond à la page Web par défaut que Delphi construit et place dans le répertoire HTML.

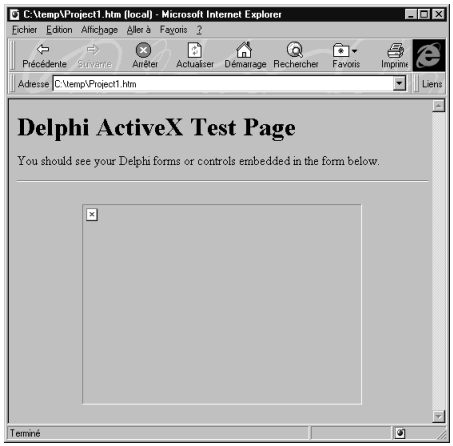
Listing 14.7 : La page Web générée par l'option Déploiement Web de Delphi

```
<HTML>
<H1> Delphi ActiveX Test Page </H1><p>
You should see your Delphi forms or controls embedded in the
form below.
<HR><center><P>
<OBJECT
  classid="clsid:E1C4AE03-5B32-11D0-9FB3-444553540000"
  codebase="http://www.coolmath.com/actform/cmproj.ocx#version=1,0,0,0"
  width=350
  height=250
  align=center
  hspace=0
  vspace=0
>
</OBJECT>
</HTML>
```

Analyse

La balise `Object` *générée indique qu'un objet doit être incorporé dans la page HTML. Les attributs* `width`, `height`, `align`, `hspace` *et* `vspace` *définissent respectivement le placement, la taille et la position du composant sur la fiche. L'attribut* `classid` *définit le composant à insérer. Si la machine dispose déjà du composant ActiveX correspondant à l'ID de classe, elle se contentera d'incorporer ce composant dans la fiche. Si celui-ci n'est pas enregistré dans le système, la balise* `codebase` *sert à le télécharger. Les informations de version garantissent que l'utilisateur dispose d'une version à jour de l'OCX. La Figure 14.9 montre la page Web créée par Delphi.*

Figure 14.9
Une page Web d'exemple créée par Déploiement Web.



Note

Si votre browser Web refuse d'accepter le composant parce qu'il n'est pas signé numériquement, vous devez modifier les paramètres de sécurité de votre browser de manière à ce que les composants ActiveX non authentifiés soient acceptés. Consultez la documentation de MSIE pour plus de détails sur les différents niveaux de sécurité.

Si vous mettez à jour la version de l'OCX et souhaitez que le browser réinstalle le composant, vous devez fermer le browser puis le relancer. Vous pouvez utiliser `-1,-1,-1,-1` *comme version pour que le composant soit toujours téléchargé. Ainsi, la ligne suivante :*

```
codebase=
"http://www.coolmath.com/actform/CMPROJ.ocx#version=1,2,0,0"
```

lancera le téléchargement de la DLL et réinstallera le composant si l'utilisateur ne dispose que de la version 1.1.0.0 ou antérieure.

Pour que votre page Web ressemble à celle de la Figure 14.7, vous devez modifier le code HTML. Vous pouvez le faire manuellement, ou utiliser à cet effet un éditeur HTML. Le Listing 14.8 montre le code HTML correspondant à la page Web de la Figure 14.7.

Listing 14.8 : Le code HTML pour la page Web de cours de maths utilisant une ActiveForm

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>

<head>
<meta http-equiv="Content-Type"
content="text/html; charset=iso-8859-1">
<meta name="GENERATOR" content="Microsoft FrontPage 2.0">
<title>Maths amusantes</title>
</head>

<body bgcolor="#FFFFFF">

<h3>Maths amusantes - Leçon 5 - Aires de cercles et de carr&eacute;s </h3>

<hr>

<p>L'aire de la forme décrit la quantité de surface contenue au sein de la
forme. On peut calculer les dimensions des cercles et des carrés à partir
d'une aire donnée en utilisant les formules suivantes :</p>

```

```
<table border="0">
    <tr>
        <td rowspan="3"><!--webbot bot="HTMLMarkup" startspan -->
        <OBJECT
        classid="clsid:E1C4AE03-5B32-11D0-9FB3-444553540000"
        codebase="http://www.coolmath.com/actform/
        CMPROJ.ocx#version=-1,-1,-1,-1"
        width=300
        height=175
        align=center
        hspace=0
        vspace=0
        >
</OBJECT><!--webbot
        bot="HTMLMarkup" endspan --></td>
       <td><strong>Pour un cercle :</strong> A = PI * (Rayon au carré) </td>
    </tr>
    <tr>
        <td><strong>Pour un carré :</strong> A = Longueur au carré </td>
    </tr>
    <tr>
        <td>Sur la gauche, vous pouvez voir un carré et un cercle. Ils ont
toujours la même aire, bien que leurs formes soient différentes. Déplacez la
barre du curseur pour modifier l'aire des formes, et les dimensions seront
automatiquement calculées.</td>
    </tr>
</table>

<hr>

<p>Fo help send mail to <a href="mailto:teachme@coolmath.com">
teachme@coolmath.com</a></p>
</body>
</html>
```

Récapitulatif

Nous avons vu lors de cette journée comment créer des applications orientées client ou serveur en Delphi. Delphi propose un cadre de développement serveur Web très puissant qui vous per-

met de concentrer vos efforts sur la logique principale de votre code, sans devoir vous préoccuper de tous les mécanismes liés à l'interaction avec le serveur. Nous avons vu comment créer une application CGI simple, puis comment utiliser le même code pour créer une application ISAPI. Le jeu de taquin nous a permis d'illustrer l'utilisation de plusieurs chemins d'accès correspondant à différentes tâches de l'application. Cet exemple a également montré les difficultés inhérentes au développement d'une application côté serveur nécessitant des informations sur les états précédents.

Les formulaires Web permettent de collecter des informations auprès de l'utilisateur. Nous avons passé en revue les fonctionnalités intégrées de Delphi qui permettent de décoder et de lire les informations passées à un serveur via de tels formulaires. Lorsqu'il est nécessaire de tirer vraiment parti des services système d'une machine, ou lorsqu'une forte interactivité est nécessaire, une application côté serveur n'est pas adaptée. ActiveX est alors une solution séduisante, puisque cette technologie permet d'incorporer à une page Web un composant qui peut réaliser tout ce qu'une application Win32 peut habituellement faire.

Le moyen le plus simple pour créer des composants ActiveX complexes consiste à utiliser les ActiveForms de Delphi. Dans notre dernier exemple, nous avons créé un didacticiel de maths qui en incorporait une afin de modifier l'aire de formes géométriques à l'aide d'une barre de curseur.

En résumé, Delphi est l'un des outils les plus adaptés pour le développement d'applications Web client et serveur.

Atelier

L'atelier vous donne trois façons de vérifier que vous avez correctement assimilé le contenu de cette section. La section Questions - Réponses vous donne des questions qu'on pose couramment, ainsi que leur réponse, la section Questionnaire vous pose des questions, dont vous trouverez les réponses en consultant l'Annexe A, et les Exercices vous permettent de mettre en pratique ce que vous venez d'apprendre. Tentez dans la mesure du possible de vous appliquer à chacune des trois sections avant de passer à la suite.

Questions - Réponses

Q Dans le cadre du développement serveur Web de Delphi, est-il possible d'utiliser des fonctionnalités ISAPI spécifiques ou d'autres cibles de serveur prises en charge ?

R Oui. Chaque cible comporte un ensemble de classes qui héritent des fonctionnalités obligatoires de la base de code générale, mais ont des fonctions spécifiques.

Q Existe-t-il un moyen simple pour enregistrer les composants ActiveX lors de la phase de test ?

R Oui. Vous pouvez utiliser les options de menu Exécuter, Recenser le serveur ActiveX et Exécuter, Dé-recenser le serveur ActiveX pour installer et désinstaller les ActiveForms au sein même de l'environnement de développement.

Q Est-il possible de lancer une application serveur Web dans le débogueur ?

R Oui. Vous devez lancer le débogueur sur votre serveur et déboguer le process de serveur Web lui-même. Pour plus de détails, consultez la documentation en ligne de Delphi.

Questionnaire

1. Les applications CGI s'exécutent-elles dans l'espace de processus du serveur Web ? Et les applications ISAPI ?
2. Lorsque vous utilisez des formulaires pour recueillir des informations d'un client Web, à quelle propriété devez-vous accéder pour décoder automatiquement les données qu'il envoye ?
3. Lorsqu'un utilisateur accède à une page Web contenant une fiche ActiveX qui n'a pas été enregistrée sur le système, comment fait son browser pour savoir où récupérer le composant ?

Exercices

1. Créez une application serveur Web qui reçoit ses entrées d'un formulaire contenant des boutons radio, des listes déroulantes et des cases à cocher.
2. Créez une application serveur Web utilisant les composants de génération HTML de base de données.
3. Convertissez le jeu de taquin en une Activeform.

A

Réponses aux questionnaires

Jour 1

1. Le premier avantage est que vous pouvez, moyennant un minimum d'efforts, effectuer un prototypage rapide de votre interface pour la montrer à vos clients. Le deuxième avantage est que vous n'êtes pas forcé d'écarter ce prototype une fois la démonstration faite, comme c'est le cas avec des outils de développement d'interface. Vous pouvez transformer votre prototype en un produit fini. Vous gagnez donc un temps précieux. Enfin, autre avantage, le RAD vous offre un ensemble d'outils de haut niveau et parfaitement intégré qui optimise votre productivité.
2. Les composants visuels sont différents des contrôles ActiveX sur bien des points. Ainsi, par exemple, les composants visuels sont compilés directement dans votre exécutable car ils sont écrits en code Delphi natif, alors que les contrôles ActiveX doivent être fournis dans des fichiers séparés de votre logiciel.
3. La mise en évidence de la syntaxe est une fonctionnalité de l'EDI de Delphi, qui fait que l'éditeur comprend le langage Delphi. Cette compréhension (limitée) lui permet à ce dernier de colorer les différentes parties de votre code (les commentaires en gris, les mots réservés en bleu). Vous pouvez modifier les couleurs utilisées. Cette mise en évidence vous permet de lire le code et de repérer ldes éventuelles erreurs plus facilement.
4. En utilisant la sélection de menu Composant | Installer un composant dans le menu principal de Delphi.

Jour 2

1. Les constantes ne peuvent pas changer de valeur en cours d'exécution de l'application, tandis que c'est le principe même des variables.
2. Les applications mathématiques dans le code doivent se comporter conformément aux lois mathématiques. Dans le cas contraire, l'ordinateur générerait des réponses fausses.
3. Les constantes typées sont en fait des variables pré-initialisées. L'avantage de ces constantes typées est que la pré-initialisation permet de fournir une valeur à vos variables dès le début du programme.

Jour 3

1. 100 fois. La boucle externe fait 10 itérations et pour chacune de ces 10 itérations, la boucle interne en effectue dix autres. Le total est donc de 10 x 10 = 100.
2. `while...do` teste la condition avant l'entrée dans la boucle.

3. Une fonction est conçue pour renvoyer une valeur unique, tandis qu'une procédure est conçue pour renvoyer un nombre indéfini de valeurs (zéro ou plus).
4. Le passage de grandes structures de données peut altérer les performances. Il est plus efficace pour la mémoire de passer plutôt un pointeur.

Jour 4

1. Les objectifs sont : possibilité de modification, efficacité, fiabilité et compréhension aisée.
2. Elle prend en charge la dissimulation de l'information en créant des unités. Les unités vous permettent de cacher les détails d'implémentation d'une application et d'offrir une interface aux données.
3. Parce que, dans le monde réel, il existe des objets qui sont basés sur d'autres objets. En Delphi, vous pouvez créer une classe *voiture* puis créer des objets qui en découlent et qui héritent des propriétés de la classe *voiture*, auxquelles peuvent s'ajouter les leurs propres. Le résultat pourrait être une classe *décapotable*, une classe *formule 1*, etc.

Jour 5

1. .PAS, .DPR, .DOF, .DFM et .RES sont créés lors de la conception. .DCU, et .EXE sont créés lors de la compilation.
2. En-tête, partie interface, partie d'implémentation, partie d'initialisation et partie de finalisation.
3. Pour ajouter une nouvelle fiche à votre projet, vous pouvez sélectionner le bouton Nouvelle fiche dans la barre d'outils ou choisir l'option de menu Fichier | Nouvelle fiche. Pour supprimer une fiche du projet, vous pouvez utiliser l'interface du Gestionnaire de projets pour sélectionner la fiche, puis cliquer sur le bouton Supprimer unité. Vous pouvez également sélectionner la fiche à supprimer et utiliser l'icône Supprimer fichier du projets qui se trouve sur la barre d'outils, ou encore l'option Fichier | Supprimer du projet.
4. Lorsque vous supprimez une fiche d'un projet, les fichiers .PAS ou .DFM associés ne sont pas supprimés, seules leurs références sont retirées du fichier source du projet.

Jour 6

1. Les options prédéterminées sont IDE classique, émulation BRIEF, et émulation epsilon.
2. Dans l'onglet Editeur de la boîte de dialogue Options d'environnement.
3. La gouttière sert à définir des points d'arrêt et à indiquer leur emplacement en affichant un point rouge. Elle permet également d'indiquer le point d'exécution du programme (flèche verte).
4. Pour déboguer une DLL sous Delphi 3, vous devez écrire une application hôte qui appellera la DLL. Sélectionnez Exécuter | Paramètres, et saisissez le nom de l'application hôte dans l'emplacement prévu à cet effet. Ajoutez points d'arrêt et de suivi à votre convenance, puis exécutez l'application hôte afin de lancer la DLL. Delphi s'arrêtera au premier point d'arrêt de votre DLL, et vous pourrez mener à bien votre session de débogage.
5. WinSight32 est un espion système, qui vous permet d'observer l'activité des process système, des messages, etc.
6. Les trois fonctionnalités d'Audit de Code sont l'Expert Modèles de code, l'Expert achèvement de code, et l'Expert paramètres de code. Le premier affiche la syntaxe des structures de code, l'achèvement de code termine les lignes à votre place en vous proposant les différentes options possibles, et l'Expert paramètres vous rappelles les paramètres et leur ordre pour toute fonction ou procédure.
7. Faux. Le compilateur suspend l'exécution avant la ligne sur laquelle est défini le point d'arrêt.
8. Pour reprendre l'exécution, supprimez les points d'arrêt et appuyez sur F9.

Jour 7

1. Le bouton droit de la souris fait apparaître les menus contextuels concernant l'objet sur lequel vous cliquez.
2. Ces fenêtres de tailles spécifiques sont adaptées à différentes résolutions d'écran. De plus, en les utilisant, vous assurez un caractère uniforme aux applications.
3. De la sorte, l'utilisateur ne perd pas de temps à apprendre à naviguer dans votre barre de menus, et peut employer ce temps à apprendre à utiliser l'application elle-même.

Jour 8

1. L'utilisateur peut visualiser un composant visuel en cours d'exécution si la propriété `visible` est définie comme `true`. Un composant non visuel est visible lors de la conception, mais pas lors de l'exécution.
2. Les propriétés imbriquées sont des propriétés placées à l'intérieur de propriétés. Vous pouvez savoir si une propriété recèle des propriétés imbriquées si un signe plus (+) figure à gauche de son nom. Si vous cliquez sur ce signe plus, il se développe pour montrer les propriétés imbriquées. Les propriétés peuvent être imbriquées sur plusieurs niveaux.
3. Une méthode est une fonction ou procédure déclarée dans un composant ou un objet, et qui peut être appelée pour affecter le comportement ou l'aspect de l'objet.
4. Les composants et les objets ont des événements qui leurs sont associés et qui, lorsqu'ils sont activés, exécutent les gestionnaires d'événements (des bouts de code) qui leur sont associés. Un clic de souris est un exemple d'événement usuel.
5. MaskEdit.
6. Timer.

Jour 9

1. Les fichiers typés sont des fichiers formatés de façon particulière et qui stockent un type de données particulier dans des champs spécifiques.
2. Les fichiers non typés sont des fichiers ne possédant nipas de format ni de type de données particuliers.
3. `Seek(var F; N : Longint);`
4. Non, vous pouvez ouvrir le port de l'imprimante comme s'il s'agissait d'un fichier et utiliser la procédure Pascal `writeln` pour envoyer du texte à l'imprimante.
5. Vous devez ajouter `Printers` dans la clause `uses`.
6. `BeginDoc` et `EndDoc`.
7. Vrai.
8. Vous n'avez pas à exécuter de code dans la boîte de dialogue Propriétés. On y accède par l'intermédiaire de la boîte de dialogue Impression.
9. `CopyRect`.
10. Il suffit de tracer le cercle sur le canevas de Printer, comme vous le feriez sur un canevas d'écran, celui d'une fiche par exemple.

```
Begin
    {on lance la tâche d'impression}
    Printer.BeginDoc;
    {On définit la largeur du crayon comme étant de 5 pixels}
    Printer.Canvas.Pen.Width:=5;
    {On dessine une ellipse dont le coin supérieur gauche se trouve à 0, et
le coin inférieur droit à 200,200}
    Printer.Canvas.Ellipse(0, 0, 200, 200);
    {on termine et imprime la tâche d'impression}
    Printer.EndDoc;
end;
```

Jour 10

1. Le triangle ne s'effacerait pas de lui-même et laisserait une trace au cours de son déplacement.
2. La façon la plus simple d'effectuer un découpage dans une région rectangulaire consiste à utiliser un composant visuel PaintBox. Le composant PaintBox empêche tout dessin hors de la zone de dessin.
3. Les composants visuels d'Image peuvent encapsuler un objet bitmap. Les objets bitmap possèdent une méthode LoadBitmap qui peut être utilisée pour charger un bitmap d'un fichier vers l'objet bitmap qui, à son tour, charge l'image dans le composant d'image. La méthode `CopyRect` peut être utilisée pour copier une partie de n'importe quel canevas (y compris un canevas dans un bitmap) dans n'importe quel autre canevas (y compris celui d'une paintbox).
4. PlaySound accepte un ensemble de paramètres qui déterminent la façon dont un son sera joué et où sera située la source du son. Un de ces paramètres détermine si le son est joué de façon synchrone ou non synchrone. Si le son est joué de façon synchrone, l'application attend que le son ait été joué. En revanche, si le son est joué de façon non synchrone, l'application continue de s'exécuter tandis que le son est joué.
5. Ce code définit une région d'affichage entre les pixels 50, 50 et 250, 250. Le premier ensemble indique la position (50, 50) et le second la largeur et la hauteur (200, 200).
6. Si vous dessinez directement sur un canevas, vous créez un effet saccadé. Pour que l'animation soit plus fluide, il faut dessiner sur un bitmap puis copier l'image dans le canevas visuel.

Jour 11

1. Une exception surviendra si vous tentez d'exécuter n'importe quelle méthode d'édition alors qu'une table est dans un état `dsBrowse`.
2. On peut accéder à des champs comme à des composants. Le composant `TField` a une propriété `DataType` qui indique le type de données contenu dans un champ particulier.
3. La propriété `IndexName` d'un `TTable` peut être défini comme un index secondaire pour ordonner les enregistrements par l'index spécifié.
4. Les bases de données en partage de fichier et client/serveur prennent toutes deux en charge l'accès simultané à la base de données. Certaines bases de données indépendantes n'autorisent l'accès qu'à une seule application à la fois.
5. Il est impossible de mettre à jour les champs non calculés lorsque l'ensemble de données est dans l'état `dsCalculate`. Seuls les champs calculés peuvent être modifiés. Cependant, tous les champs peuvent être consultés.
6. Un champ indiquant une clé dans une table différente est qualifié de clé étrangère.

Jour 12

1. QuickReport est un générateur d'états par bandes.
2. Les bandes sont utilisées pour afficher des données, des images, et servent de conteneur pout d'autres composants QuickReport afin de construire visuellement l'état.
3. Non. Les composants QuickReport sont destinés à concevoir et à créer l'état, pas à être vus de l'utilisateur.
4. Un `TTable` se connecte la à la base de données, et le composant `TQuickRep` relie les bandes et composants QuickReport à la table.

Jour 13

1. Lorsque l'utilisateur d'un composant définit une propriété, cela peut déclencher une procédure qui traite tous les mécanismes en jeu.
2. Les propriétés disponibles lors de la conception peuvent être définies en utilisant l'Inspecteur d'objets. Les propriétés disponibles lors de l'exécution ne peuvent être définies que par le biais de code Pascal.

3. Une propriété publique est disponible à tout code lors de l'exécution. En spécifiant qu'une propriété est publiée, vous la faites apparaître dans l'Inspecteur d'objet et la rendez ainsi disponible lors de la conception.
4. Propriétés et événements sont déclarés de façon identique, mais les événements sont définis comme des types de propriétés spéciales qui font que l'Inspecteur les fait figurer dans la page Evénements.
5. Un ID de classe est l'identifiant unique qui permet de distinguer une classe donnée de toutes les autres. Une application peut instancier une classe en utilsant son ID de classe. Une bibliothèque de types définit l'ensemble des fonctions d'interface. Une application peut utiliser une bibliothèque de types pour accéder aux fonctions d'un objet COM avant compilation ou exécution.
6. Oui, les composants ActiveX peuvent être utilisés dans toute application ou langage sachant créer une instance d'un composant ActiveX. Internet Explorer, Visual Basic, Visual C++ et Microsoft Word sont des exemples d'applications pouvant utiliser des composants ActiveX.

Jour 14

1. Les applications CGI tournent dans leur propre espace mémoire. Dès lors, une application CGI erronée ne peut pas faire planter le serveur Web. D'un autre côté, il est coûteux en ressources système de relancer un process pour chaque nouvelle requête. Une application ISAPI tourne dans le même espace mémoire que le serveur Web, ce qui accroît les performances et les risques d'instabilité.
2. La propriété `TWebRequest.QueryString` permet de lire les informations décodées transmises par le client. Vous gagnez ainsi du temps, puisque vous n'avez pas à écrire vous-même les routines de décodage.
3. La balise `codebase` sert à indiquer au client Web l'emplacement du contrôle ActiveX s'il n'est pas déjà installé sur la machine de l'utilisateur. On peut ajouter des informations de version pour s'assurer que l'utilisateur dispose de la bonne version du contrôle

Index

A

B

C

D

E

F

G

H

N

O

P

Q

R

S

T

U

V

W