Learning Objects for Java (with Jeliot)

By:
Mordechai (Moti) Ben-Ari

Learning Objects for Java (with Jeliot)

By:
Mordechai (Moti) Ben-Ari

Online:
< http://cnx.org/content/col10915/1.2/ >

C O N N E X I O N S

Rice University, Houston, Texas

Collection structure revised: December 28, 2009
PDF generated: February 5, 2011
For copyright and attribution information for the modules contained in this collection, see p. 69.

Table of Contents

Chapter 1

Learning Objects for Java (Overview)[1]

Overview:

Learning objects (LOs) are small, self-contained, reusable resources for learning. The advantages of LOs include: *flexibility of use* (students can choose to work with LOs at their convenience) and *adaptability* (students can choose to work only with those LOs that address topics they find difficult).

This collection contains five modules, each with about ten LOs for the topic of the module: control structures, arrays, methods, constructors, inheritance.

The LOs in this collection are designed for use with the Jeliot system for animating introductory programs in Java.

A learning object consists of text and Java programs. Each LO is independent, so if you know the needed background material you can go directly to any LO. For each topic, a table is given that lists the LOs, the associated source files, and the "prerequisites" for each LO. The prerequisites are the number of the LO that introduces *concepts* that are assumed; however, there is no need to actually work through the LOs in sequence.

The text for each LO starts with a description of the concept being presented and an overview of the example program. It is followed by a bulleted list for each program that describes what to observe as you *step* through the program with Jeliot. The text for the LO ends with a programming exercise.

Installation:

Before you begin, download and install Jeliot from the link given in the sidebar. Download the zip files with the source code for each of the LOs. There is a zip file associated with each module that contains the source files for the LOs for its topic; in addition, there is a zip file learning-objects.zip[2] with the source files for all the LOs in the collection.

Tips for using Jeliot:

- The LOs have been tested with Jeliot Version 3.7.1; please ensure that you are not using earlier versions.
- Copy the source file directories to a clean directory so that if you make changes you will not modify the original files. Run Jeliot and open the source file for the LO you want to work with.
- Learn how to use Jeliot before studying the LOs. In particular, learn how to use `Step`, `Pause`, `Play`, and `Rewind` to control the animation.
- Select `Animation / Run Until... (ctrl-T)` and enter a line number to begin the animation at that line. This is very useful in two situations:
    - when you are animating a program several times and wish to skip over the initialization or other parts of the code;

[1] This content is available online at <http://cnx.org/content/m31242/1.3/>.
[2] See the file at <http://cnx.org/content/m31242/latest/learning-objects.zip>

  - when you wish to examine the final state after the last line of the main method: enter the line number of the closing brace of the main method.

- Select `Options / Show History View` to enable storing of each step of the animation; these can be viewed by selecting the `History` tab on the right-hand side of the display. Enabling the history may slow Jeliot down, especially for large programs.
- The programs in the LOs use standard Java with two exceptions that simplify the animations:
  - None of the programs use the parameter of the `main` method. Since Jeliot accepts Java programs without the formal parameter definition `String[] args`, the parameter has been commented-out in the programs. You can remove the comments to compile the programs with a Java compiler. If you wish to run Jeliot with the parameter, you can select `Options / Use Null Parameter to Call Main` to skip over the animation of the parameter.
  - Two of the LOs on control structures use the input statement: `input = Input.nextInt()`. To compile these programs with standard Java, add the following declaration to the `main` method:

```
java.util.Scanner~Input~=~new~java.util.Scanner(System.in);
```

Acknowledgements:
I would like to thank Niko Myller and Andrés Moreno for modifying Jeliot to accomodate the LOs, and Ronit Ben-Bassat Levy for suggestions for improving the LOs.

Chapter 2

Learning Objects for Control Structures in Java[1]

2.1 Learning Objects for Control Statements

Concept Normally, statements in Java are executed sequentially in the order written in the source code. *Control statements* are used to modify the order of execution of statements. Most control statements are *conditional*; that is, the next statement to be executed depends on the result of evaluating an expression, usually, an expression that returns a *boolean* value of true or false.

These source code of these learning objects can be found in control.zip[2].

LO	Topic	Java Files (.java)	Prerequisites
"If-statements" (Section 2.1.1: If-statements)	If-statements	Control01	
"Conditional expressions" (Section 2.1.2: Conditional expressions)	Conditional expressions	Control02	1
"While loops" (Section 2.1.3: While loops)	While loops	Control03	
"Do-while loops" (Section 2.1.4: Do-while loops)	Do-while loops	Control04	

continued on next page

[1]This content is available online at <http://cnx.org/content/m31246/1.1/>.
[2]See the file at <http://cnx.org/content/m31246/latest/control.zip>

"Break statements" (Section 2.1.5: Break statements)	Break statements	Control05	3
"Counting with for statements" (Section 2.1.6: Counting with for statements)	Counting with for statements	Control06	
"General for statements" (Section 2.1.7: General for statements)	General for statements	Control07	6
"Continue statements" (Section 2.1.8: Continue statements)	Continue statements	Control08	6
"Switch statements" (Section 2.1.9: Switch statements)	Switch statements	Control09	

Table 2.1

2.1.1 If-statements

Concept The execution of an if-statement starts with the evaluation of its boolean-valued expression. If the result is true, the statement written after the closing parenthesis of the expression is executed; if the result is false, the statement written after the `else` is executed. These statements can be single statements or blocks of statements. In particular, the statements can themselves be if-statements (*nested if-statements*), in which case the inner statement is executed the same way.

Program: Control01.java

```
    //~Learning~Object~Control01
//~~~~if~statements
public~class~Control01~{
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~int~year~=~2000;
~~~~~~~~int~month~=~6;
~~~~~~~~int~days;
~~~~~~~~if~(month~==~2)
~~~~~~~~~~~~if~(year~%~4~==~0)
~~~~~~~~~~~~~~~~days~=~28;
~~~~~~~~~~~~else
~~~~~~~~~~~~~~~~days~=~29;
~~~~~~~~else~if~(month~==~4~||~month~==~6~||
~~~~~~~~~~~~~~~~month~==~9~||~month~==~11)
~~~~~~~~~~~~days~=~30;
~~~~~~~~else
~~~~~~~~~~~~days~=~31;
~~~~~~~~System.out.println(days);
~~~~}
}
```

The program computes the number of days in a month taking leap years into account.

- The variables are allocated and the first two, `year` and `month`, are given initial values.
- The expression `month == 2` evaluates to false, so the statement following the else is executed. Jeliot will display `Choosing else-branch` to emphasize this.
- The inner statement is itself an if-statement. The expression is evaluated and its result is true. Note that once one of the terms of `||` (or) becomes true, there is no need to evaluate the others.
- The assignment statement following the statement is executed. Jeliot will display `Choosing then-branch`. (The terminology *then-branch* orginates from programming languages that require the use of the keyword `then` between the expression and the statement.)
- The value of `days` is printed.

Exercise Complete the program with the correct computation for leap years: a year divisible by 100 is not a leap year unless it is also divisible by 400.

2.1.2 Conditional expressions

Concept A conditional expression is a shorthand for an if-statement that assigns different values to one variable:

```
if (expression) var = value1; else var = value2;
```

This can be rewritten more concisely as:

```
var = (expression) ? value1 : value2;
```

The boolean-valued expression is evaluated: If the result is true, `value1` is assigned to `var`; if not, `value2` is assign to `var`.

Program: Control02.java

```
    //~Learning~Object~Control02
//~~~~conditional~expressions
public~class~Control02~{
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~int~year~=~2001;
~~~~~~~~int~month~=~2;
~~~~~~~~int~days;
~~~~~~~~if~(month~==~2)
~~~~~~~~~~~~days~=~(year~%~4~==~0)~?~28~:~29;
~~~~~~~~else~if~(month~==~4~||~month~==~6~||
~~~~~~~~~~~~~~~~month~==~9~||~month~==~11)
~~~~~~~~~~~~days~=~30;
~~~~~~~~else
~~~~~~~~~~~~days~=~31;
~~~~~~~~System.out.println(days);
~~~~}
}
```

The program computes the number of days in a month taking leap years into account.

- The variables are allocated and the first two, `year` and `month`, are given initial values.
- The expression `month == 2` evaluates to true, so the statement following the expression is executed. Jeliot will display `Choosing then-branch` to emphasize this.
- The inner statement is an assignment statement with a conditional expression. The expression is evaluated and its result is false, so the value after the colon is assigned to the variable. Jeliot will display `Choosing else-branch`.

- The value of `days` is printed.

Exercise Complete the program with the correct computation for leap years: a year divisible by 100 is not a leap year unless it is divisible by 400.

Exercise Rewrite the entire if-statement as nested conditional expressions.

2.1.3 While loops

Concept A loop enables the execution of a statement (including a block of statements within braces) an arbitrary number of times. This statement is called the *loop body*. In a while loop, an expression is evaluated *before* each execution of the loop body, and loop body is executed if and only if the expression evaluates to true.

Program: Control03.java

```
    //~Learning~Object~Control03
//~~~~while~loops
public~class~Control03~{
~~~~static~int~LIMIT~=~100;
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~int~factorial~=~1;
~~~~~~~~int~n~=~1;
~~~~~~~~while~(factorial~<~LIMIT)~{
~~~~~~~~~~~~System.out.println(factorial);
~~~~~~~~~~~~n++;
~~~~~~~~~~~~factorial~=~factorial~*~n;
~~~~~~~~}
~~~~}
}
```

This program prints all factorials less than `LIMIT = 100`, namely, $1! = 1$, $2! = 2$, $3! = 6$, $4! = 24$.

- The static constant and the two variables are allocated and initialized.
- Then, and each time the keyword `while` is reached, the expression is evaluated. If it is true, execution proceeds with the loop body, and Jeliot displays `Entering the while loop` the first time and `Continuing the while loop` on subsequent occasions.
- The statements of the loop body are executed. They print the value of the current factorial, increment the counter and compute the new factorial; then, control returns to the while-expression.
- If and when the expression evaluates to false, execution proceeds with the statement following the loop body. Jeliot displays `Exiting the while loop`.

Exercise According to a formula by Euler,

$$\frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \cdots = \frac{\pi^2}{6} \tag{2.1}$$

Write a program to compute the series until the difference between the two terms is less than 0.1.

2.1.4 Do-while loops

Concept A loop enables the execution of a statement (including a block of statements within braces) an arbitrary number of times. This statement is called the *loop body*. In a do-while loop, an expression is evaluated *after* each execution of the loop body, and the loop body continues to execute if and only if the expression evaluates to true.

The loop body of a do-while loop will execute at least one time. This type of statement is particularly appropriate for processing input, because you need to input data at least once before you can test it in an expression.

Program: Control04.java

```
    //~Learning~Object~Control04
//~~~~do-while~loops
public~class~Control04~{
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~int~input;
~~~~~~~~do~{
~~~~~~~~~~~~~input~=~Input.nextInt();
~~~~~~~~}~while~(input~<=~0);
~~~~~~~~System.out.println(input);
~~~~}
}
```

The program reads interactive input until a positive number is entered.

- The variable `input` is allocated but not initialized.
- The loop body of the do-while loop is executed. Jeliot displays `Entering the do-while loop`.
- A value is read interactively into the variable `input`. First, enter a negative integer.
- The expression following the `while` is evaluated. Since it evaluates to true, the loop body is executed again. Jeliot displays `Continuing the do-while loop`.
- Now enter a positive value into the variable `input`.
- The expression following the `while` is evaluated. Since it evaluates to false, the execution of the do-while loop is completed. Jeliot displays `Exiting the do-while loop`.

Exercise Rewrite this program with a while loop. Compare it to the do-while loop.

2.1.5 Break statements

Concept The exit from a while loop occurs *before* the loop body and the exit from a do-while loop occurs *after* the loop body. The `break` statement can be used to exit from an arbitrary location or locations from within the loop body.

The `break` statement is useful when the expression that leads to exiting the loop cannot be evaluated until some statements from the loop body have been executed, and yet there remain statements to be executed after the expression is evaluated.

Program: Control05.java

```
    //~Learning~Object~Control05
//~~~~break~statements
public~class~Control05~{
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~int~input;
~~~~~~~~int~sum~=~0;
~~~~~~~~while~(true)~{
~~~~~~~~~~~~input~=~Input.nextInt();
~~~~~~~~~~~~if~(input~<~0)~break;
~~~~~~~~~~~~sum~=~sum~+~input;
~~~~~~~~}
~~~~~~~~System.out.println(sum);
~~~~}
}
```

The program sums a sequence of nonnegative integers read from the input and terminates when a negative value is read.

- The two variables are allocated and `sum` is initialized with the value zero.
- The `while` statement is executed with `true` as the loop expression. Of course, `true` will never evaluate to false, so the loop will never be exited at the `while`.
- An integer value is read from the input. If it is negative the `break` statement is executed and Jeliot displays `Exiting the while loop because of the break`.
- Otherwise, the following assignment statement is executed and Jeliot displays `Continuing without branching`.
- After the assignment statement is executed, the loop starts again.

Exercise Write equivalent programs using a while loop and a do-while loop.

2.1.6 Counting with for statements

Concept Although all loop structures can be programmed as `while` loops, one special case is directly supported: writing a loop that executes a predetermined number of times. The `for` statement has three parts:

```
for (int i = 0; i < N; i++)
```

The first part declares a loop control variable and gives it an initial value. The second part contains the exit condition: the loop body will be executed as long as the expression evaluates to true. The third part describes how the value of the control variable is modified after executing the loop body. The syntax show is the conventional one for executing a loop `N` times.

Program: Control06.java

```
//~Learning~Object~Control06
//~~~~counting~with~for~statements
public~class~Control06~{
~~~~static~final~int~N~=~6;
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~int~factorial~=~1;
~~~~~~~~for~(int~i~=~0;~i~<~N;~i++)
~~~~~~~~~~~~factorial~=~factorial~*~(i+1);
~~~~~~~~System.out.println(factorial);
~~~~}
}
```

This program computes the first six factorials in a `for` loop and the last value is printed.

- The constant `N` and the variable `factorial` are allocated and initialized.
- The control variable `i` is allocated and initialized.
- The expression `i < N` is evaluated and evaluates to true. Jeliot displays `Entering the for loop`.
- The loop body is executed.
- The control variable is incremented as specified in the third part of the `for` statement.
- The previous three steps are repeated until the expression evaluates to false; this causes the loop to be exited. Jeliot displays `Continuing the for loop` as long as the expression evaluates to true, and `Exiting the for loop` when it evaluates to false.
- The final value of `factorial` is printed.
- **Important**: when the loop is exited, the control variable is deallocated and no longer exists.

Exercise Rewrite the program using a `while` loop.

2.1.7 General for statements

Concept Arbitrary expressions can be given for the initial value of the `for` statement, the exit condition, and the modification of the control variable.

Program: Control07.java

```
    //~Learning~Object~Control07
//~~~~General~for~statements
public~class~Control07~{
~~~~static~final~int~N~=~100;
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~int~sum~=~0;
~~~~~~~~for~(int~i~=~0;~i~<~Math.sqrt(N);~i~=~i~+~3)
~~~~~~~~~~~~sum~=~sum~+~i;
~~~~~~~~System.out.println(sum);
~~~~}
}
```

This program computes the sum of multiples of three that are less than the square root of `N`.

- The constant `N` and the variable `sum` are allocated and initialized.
- The control variable `i` is allocated and initialized.
- The expression `i < Math.sqrt(N)` is evaluated and evaluates to true. Jeliot displays `Entering the for loop`.
- The loop body is executed.
- The control variable is incremented by three as specified in the third part of the `for` statement.
- The previous three steps are repeated until the expression evaluates to false; this causes the loop to be exited. Jeliot displays `Continuing the for loop` as long as the expression evaluates to true, and `Exiting the for loop` when it evaluates to false.
- The final value of `sum` is printed.
- **Important**: when the loop is exited, the control variable is deallocated and no longer exists.

Exercise Is `for (;;;)` legal? If so, what does it mean?

Exercise Modify the program so that the square root is computed only once.

2.1.8 Continue statements

Concept The `break` statement is used to *exit* a loop from an arbitrary location in its body; the `continue` statement is used to *skip* the rest of a loop body and return to evaluate the condition for continuing the loop.

Program: Control08.java

```
    //~Learning~Object~Control08
//~~~~continue~statements
public~class~Control08~{
~~~~static~final~int~N~=~10;
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~int~sum~=~0;
~~~~~~~~for~(int~i~=~0;~i~<~N;~i++)~{
~~~~~~~~~~~~if~(i~%~2~==~0)~{
~~~~~~~~~~~~~~~~if~(i~%~3~==~0)
~~~~~~~~~~~~~~~~~~~~continue;
~~~~~~~~~~~~~~~~else
~~~~~~~~~~~~~~~~~~~~sum~=~sum~+~i;
```

```
~~~~~~~~~~~~}
~~~~~~~~~~~~else~if~(i~%~3~==~0)
~~~~~~~~~~~~~~~~sum~=~sum~+~i;
~~~~~~~~~~~~else
~~~~~~~~~~~~~~~~continue;
~~~~~~~~}
~~~~~~~~System.out.println(sum);
~~~~}
}
```

This program sums all the positive integers less than `N` that are divisible by 2 or 3 but not by both. For `N=10`, the result is $2 + 3 + 4 + 8 + 9 = 26$.

- The constant `N` and the variable `sum` are allocated and initialized.
- The `for` loop is standard and is executed for the values 0 through $N - 1$.
- If `i` is divisible by 2 and also by 3 (for example, 6), the `continue` statement is executed and the variable `sum` is not modified.
- If `i` is divisible neither by 2 nor by 3 (for example, 5), the `continue` statement is executed and the variable `sum` is not modified.
- In all other cases, the value of `i` is added to `sum`.
- The final value of `sum` is printed.

Exercise Modify the program so that it explicitly checks for divisibility by 6, instead of checking for divisibility by 2 and 3 in separate statements.

Exercise Modify the program so that `continue` is not used.

2.1.9 Switch statements

Concept A `switch` statement is a generalization of an `if` statement. Instead of selecting between two alternatives depending on the value of a boolean-valued expression, an integer-valued expression is used and there can be multiple alternatives introduced by the keyword `case`. Since there are a very large number of integer values, an alternative labeled `default` is executed when the value in the expression is not explicitly listed in one of the alternatives.

Important: In an `if`-statement, the end of the statement (or block of statements) of the first alternative causes a transfer of control to the end of the `if` statement, skipping over the statement (or block of statements) in the second (`else`) alternative. This *does not* happen in a `switch`: control "drops through" from the end of one alternative to the beginning of the next alternative. A `break` statement must be used to transfer control from the end of an alternative to the end of the `switch` statement.

Program: Control09.java

```
    //~Learning~Object~Control09
//~~~~switch~statements
public~class~Control09~{
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~int~year~=~2001;
~~~~~~~~int~month~=~4;
~~~~~~~~int~days;
~~~~~~~~switch~(month)~{
~~~~~~~~~~~~case~2:
~~~~~~~~~~~~~~~~days~=~(year~%~4~==~0)~?~28~:~29;
~~~~~~~~~~~~~~~~break;
~~~~~~~~~~~~case~4:
~~~~~~~~~~~~case~6:
```

```
~~~~~~~~~~~~case~9:
~~~~~~~~~~~~case~11:
~~~~~~~~~~~~~~~~days~=~30;
~~~~~~~~~~~~default:
~~~~~~~~~~~~~~~~days~=~31;
~~~~~~~~}
~~~~~~~~System.out.println(days);
~~~~}
}
```

This program computes the number of days in a month.

- The variables are allocated and the first two, `year` and `month`, are given initial values.
- The switch statement chooses a `case` depending on the value of the variable `month`. Jeliot displays `Entering a switch statement`.
- The `case` associated with 4 is selected. Jeliot displays `This case is selected`. The assignment statement assigns 30 to `days`.
- The assignment statement assigns 31 to `days`.
- The switch statement terminates and Jeliot displays `Exiting a switch statement`.
- The value of `days` is printed.

Exercise Explain why the second assignment statement is executed; fix the program.

Exercise Explain why the sequence of `case`'s for $4, 6, 9, 11$ works.

Exercise Modify the program so that the `case`'s for the 31-day months are given explicitly and so that the days are computed correctly in leap years.

Chapter 3

Learning Objects for Methods in Java[1]

3.1 Learning Objects for Methods

Concept *Methods* are the simplest construct for abstraction in Java. A method starts with a declaration that defines its *signature*: the name of the method, the number and types of the *formal parameters* and the *return type*. The body of the method consists of local variable declarations and of statements. A method is *called* or *invoked* by writing the name of the method followed by a list of values, called *actual parameters*, one for each formal parameter. A method can return a value or it can be declared as `void` if no value is returned.

These source code of these learning objects can be found in method.zip[2].

LO	Topic	Java Files (.java)	Prerequisites
"A void method" (Section 3.1.1: A void method)	A void method	Method01	
"A method returning a value" (Section 3.1.2: A method returning a value)	A method returning a value	Method02	
"Calling one method from another" (Section 3.1.3: Calling one method from another)	Calling one method from another	Method03	1, 2
"Recursion" (Section 3.1.4: Recursion)	Recursion	Method04	2
			continued on next page

[1]This content is available online at <http://cnx.org/content/m31247/1.1/>.
[2]See the file at <http://cnx.org/content/m31247/latest/method.zip>

"Calling methods on an object" (Section 3.1.5: Calling methods on an object)	Calling methods on an object	Method05	2, *
"Calling a method on the same object" (Section 3.1.6: Calling a method on the same object)	Calling a method on the same object	Method06	5, *
"Objects as parameters" (Section 3.1.7: Objects as parameters)	Objects as parameters	Method07	5, *
"Returning objects" (Section 3.1.8: Returning objects)	Returning objects	Method08	7, *
"Returning locally instantiated objects" (Section 3.1.9: Returning locally instantiated objects)	Returning locally instantiated objects	Method09	8, *

Table 3.1

* This LO assumes knowledge of the declaration of classes and the instantiation of objects.

3.1.1 A void method

Concept When a method that is declared `void` is called, it allocates memory for its parameters and local variables, executes its statements and then returns. The call is a statement constructed from the name of the method followed by a list of actual parameters.

Program: Method01.java

```
    //~Learning~Object~Method01
//~~~~void~methods
public~class~Method01~{
~~~~static~void~printMax(int~a,~int~b)~{
~~~~~~~~int~max;
~~~~~~~~if~(a~>~b)
~~~~~~~~~~~~max~=~a;
~~~~~~~~else
~~~~~~~~~~~~max~=~b;
~~~~~~~~System.out.println(max);
~~~~}
~
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~int~x~=~10,~y~=~20;
~~~~~~~~printMax(x,~y);
~~~~~~~~Method01.printMax(10,~y);
~~~~}
}
```

The program computes the maximum of two integer values.

- The variables `x` and `y` are allocated and initialized.
- The method is called with the values of the actual parameters `x` and `y`.
- Memory is allocated for the formal parameters of the method and the local variables. This is called an *activation record* and is displayed by Jeliot in the upper left hand part of the screen labeled `Method Area`. The new activation record hides the previous ones which are no longer accessible.
- The actual parameters are used to initialize the formal parameters in the activation record.
- The local variable `max` is allocated within the activation record.
- The statements of the method are executed.
- After the last statement has been executed, the method *returns* and the activation record is deallocated.

- Execution continues with the statement after the method call. Here, the method is called again, this time with an integer literal as an actual parameter instead of a variable.

Note: In a call to a static method, the name of the class in which it is defined can be given as in the second call. Since the method is defined in the *same* class as the call, the class name need not be given, as shown in the first call.

Exercise Trace the execution of a call of the following method and explain why it doesn't swap the values of the actual parameters.

```
    void~swap(int~a,~int~b)~{
~int~temp~=~a;
~a~=~b;
~b~=~temp;
}
```

Can you write a method to swap two integer values?

3.1.2 A method returning a value

Concept When a method that is declared with a return type is called, it allocates memory for its parameters and local variables, executes its statements and then returns a value of the type. The call is a statement constructed from the name of the method followed by a list of actual parameters; the call is an expression and can appear wherever an expression is allowed.

Program: Method02.java

```
    //~Learning~Object~Method02
//~~~~methods~returning~a~value
public~class~Method02~{
~~~~static~int~maximum(int~a,~int~b)~{
~~~~~~~~if~(a~>~b)
~~~~~~~~~~~~return~a;
~~~~~~~~else
~~~~~~~~~~~~return~b;
~~~~}
~
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~int~x~=~10,~y~=~20;
~~~~~~~~int~max;
~~~~~~~~max~=~maximum(x,~y);
~~~~~~~~System.out.println(max);
~~~~}
}
```

- The variables `x` and `y` are allocated and initialized; the variable `max` is allocated but not initialized.
- An assignment statement is executed: the expression on the right hand side is a method call including the values of the actual parameters `x` and `y`.
- Memory is allocated for the formal parameters of the method and the local variables. This is called an *activation record* and is displayed by Jeliot in the upper left hand part of the screen labeled `Method Area`. The new activation record hides the previous ones which are no longer accessible.
- The actual parameters are used to initialize the formal parameters in the activation record.
- The statements of the method are executed.
- When the statement `return b` is executed, the value of `b` is used for the value to be returned.
- The method *returns* and the activation record is deallocated.
- The value returned becomes the value of the expression assigned to the variable `max`.
- The value of `max` is printed.

Exercise Write the body of the main method as one statement.

3.1.3 Calling one method from another

Concept One method can call another, that is, when executing one method, any statement or expression call be a method call. A sequence of method calls results in a *stack* of activation records, where each method (except the last one that was called) is waiting for the method it called to return. There is no limit on the *depth* of method calls, except of course the amount of memory allocated to the program.

Note: The `main` method is a method like any other. The operating system can be considered as a program which calls the main method. This call has a single parameter: an array of strings containing the contents of the command line.

Program: Method03.java

```
//~Learning~Object~Method03
//~~~~calling~a~method~from~a~method
public~class~Method03~{
~~~~static~int~maximum(int~a,~int~b)~{
~~~~~~~~if~(a~>~b)
~~~~~~~~~~~~return~a;
~~~~~~~~else
~~~~~~~~~~~~return~b;
~~~~}
~
~~~~static~void~printMax(int~a,~int~b)~{
~~~~~~~~int~max;
~~~~~~~~max~=~Method03.maximum(a,~b);
~~~~~~~~System.out.println(max);
~~~~}
~
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~printMax(10,~20);
~~~~}
}
```

- The `main` method calls the method `printMax`; the actual parameters are two integer literals.
- The activation record for `printMax` is allocated, and the actual parameters are used to initialize the formal parameters `a` and `b`.
- The variable `max` is allocated but not initialized.

- The method `maximum` is called; the actual parameters are the values of `a` and `b`, which are the formal variables of method `printMax`.
- An activation record is allocated for `maximum`. (There are now three activation in the stack.) The new activation record includes memory for the formal parameters `a` and `b`; note that these are new parameters not at all related to the formal parameters of the same names in the previous method `printMax` because those parameters are hidden.
- The method `maximum` executes its body and returns a value. Just before it returns, select the tab `Call Tree` above the graphic display; the sequence of calls from `main` to `printMax` and then `maximum` is displayed. Select `Theater` to return to the animated display.
- When the method returns, its activation record is deallocated, uncovering the activation record of `printMax`.
- The value returned is assigned to the variable `max` and printed.
- When `printMax` completes its execution, its activation record is deallocated.

Note: In a call to a static method, the name of the class in which it is defined can be given as in the call to `maximum`. Since the method is defined in the same class as the call, the class name need not be given, as shown in the call to `printMax`.

Exercise Write a program to compute the maximum of six values using as few statements as possible.

3.1.4 Recursion

Concept*Recursion* occurs when method calls itself. There is nothing at all mysterious about recursion! Each call simply creates a new activation record on the stack. However, to ensure that the recursive calls terminate, eventually, some call of the method should return without invoking itself once again.

Program: Method04.java

```
    //~Learning~Object~Method04
//~~~~recursion
public~class~Method04~{
~~~~static~int~factorial~(int~n)~{
~~~~~~~~if~(n~<=~1)
~~~~~~~~~~~~return~1;
~~~~~~~~else
~~~~~~~~~~~~return~n~*~factorial(n-1);
~~~~}
~
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~System.out.println(factorial(5));
~~~~}
}
```

The standard example of a recursive method is one that computes the factorial function:

$$n! = n \cdot (n-1) \cdot \dots \cdot 2 \cdot 1 = n \cdot (n-1)! \tag{3.1}$$

The recursion is terminated by defining $n! = 1$ for $n \leq 1$.

- The `main` method calls the method `factorial` with the actual parameter 5. This creates an activation record with the formal parameter `n` initialized to 5.
- To compute the expression in the second return statement, the method `factorial` is called again, this time with the actual parameter equal to $5 - 1 = 4$.
- The sequence of recursive calls continues five times, each one allocating a new activation record with a new variable `n`.

- Finally, `factorial` is called with actual parameter 1. This call creates a new activation record as usual, but does not cause `factorial` to be invoked again. Instead, the value 1 is returned and the activation record is deallocated. Just before the method returns, select the tab `Call Tree` above the graphic display; the sequence of calls from `main` to the sequence of recursive calls is displayed. Select `Theater` to return to the animated display.
- The recursive sequence *unfolds*: each returned value is used to compute a new value to be returned by that call of `factorial`.
- Finally, the value 120 is returned to the `main` method and printed.

Exercise Write a recursive method to compute the n'th Fibonacci number:

$$fib\,(0) = 0, fib\,(1) = 1, fib\,(n) = fib\,(n-1) + fib\,(n-2) \quad for \;\; n > 1. \tag{3.2}$$

Exercise Write a more efficient nonrecursive method for the same function.

3.1.5 Calling methods on an object

Concept Nonstatic methods defined in a class must be invoked *on* an object of the class. A reference to the object becomes an *implicit* actual parameter that initializes a formal variable called `this` in the method. The variable `this` need not be explicitly mentioned when accessing fields of the object unless there is an ambiguity.

Program: Method05.java

```
    //~Learning~Object~Method05
//~~~~calling~methods~on~an~object
class~Song~{
~~~~int~seconds;
~
~~~~Song(int~s)~{
~~~~~~~~seconds~=~s;
~~~~}
~
~~~~double~computePrice(double~pricePerSecond)~{
~~~~~~~return~seconds~*~pricePerSecond;
~~~~}
}
~
public~class~Method05~{
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~Song~song1~=~new~Song(164);
~~~~~~~~Song~song2~=~new~Song(103);
~~~~~~~~double~price1~=~song1.computePrice(0.01);
~~~~~~~~double~price2~=~song2.computePrice(0.02);
~~~~~~~~System.out.println(price1);
~~~~~~~~System.out.println(price2);
~~~~}
}
```

This program computes the cost of a song as the product of its length in seconds and the price per second. A class `Song` is defined to encapsulate the field `seconds` and the method `computePrice`.

- Two objects of class Song are instantiated and references to them are assigned to the variables `song1` and `song2`.

- The method `computePrice` is called *on* the object referenced by `song1`. In Jeliot this is visualized by an arrow to the object placed in the `Expression Evaluation Area` followed by a period and the method name and parameters.
- An activation record is allocated containing two formal parameters: `this` is initialized by the implicit reference and `pricePerSecond` is initialized from the actual parameter.
- The reference in the parameter `this` is used to obtain the value of the field `seconds`. An expression is evaluated and its value returned.
- The activation record is deallocated and the value returned is stored in the variable `price1`.
- A second call to the method is executed in exactly the same way, except that it is called *on* the object referenced by `song2`.
- The values of `price1` and `price2` are printed.

Exercise Modify the method so that the formal parameter is *also* named `seconds`. Yes, it can be done! (Hint: read the **Concept** paragraph above.)

3.1.6 Calling a method on the same object

Concept A nonstatic method defined in a class that is invoked *on* an object of the class can invoke another such method on the same object. The object for the second call is the same as the one on the first call, namely, the only referenced by `this`. There is no need to explicitly write `this` and the object may be accessed implicitly.

Program: Method06A.java

```
    //~Learning~Object~Method06A
//~~~~calling~a~method~on~the~same~object
class~Song~{
~~~~int~seconds;
~
~~~~Song(int~s)~{
~~~~~~~~seconds~=~s;
~~~~}
~
~~~~boolean~discount(int~s)~{
~~~~~~~~return~s~>~300;
~~~~}
~
~~~~double~computePrice(double~pricePerSecond)~{
~~~~~~~~double~price~=~seconds~*~pricePerSecond;
~~~~~~~~if~(discount(seconds))
~~~~~~~~~~~~price~=~price~*~0.9;
~~~~~~~~return~price;
~~~~}
}
~
public~class~Method06A~{
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~Song~song1~=~new~Song(164);
~~~~~~~~Song~song2~=~new~Song(403);
~~~~~~~~double~price1~=~song1.computePrice(0.01);
~~~~~~~~double~price2~=~song2.computePrice(0.01);
~~~~~~~~System.out.println(price1);
~~~~~~~~System.out.println(price2);
```

```
~~~~}
}
```

This program computes the cost of a song as the product of its length in seconds and the price per second. A discount is applied to "long" songs. A class `Song` is defined to encapsulate the field `seconds` and the methods `computePrice` and `discount`.

- Two objects of class Song are instantiated and references to them are assigned to the variables `song1` and `song2`.
- The method `computePrice` is called *on* the object referenced by `song1`. In Jeliot this is visualized by an arrow to the object placed in the `Expression Evaluation Area` followed by a period and the method name and parameters.
- An activation record is allocated containing two formal parameters: `this` is initialized by the implicit reference and `pricePerSecond` is initialized from the actual parameter.
- The local variable `price` is declared and initialized by the expression calculated from the formal parameter `pricePerSecond` and the field of the object `seconds` that is implicitly accessed through `this`.
- The method `discount`, declared in the same class, is invoked and returns a boolean value. A new activation is allocated for this method and deallocated when it terminates. The implicit actual parameter is `this` and it is used to initialize the implicit formal parameter `this` of the method `discount`.
- The activation record for `computePrice` is deallocated and the value returned is stored in the variable `price1`.
- A second call to the method is executed exactly the same way, except that it is called *on* the object referenced by `song2`.
- The values of `price1` and `price2` are printed.

Exercise Modify the program so that discount does not use the explicit parameter `s`.

Program: Method06B.java

```
//~Learning~Object~Method06B
//~~~~calling~a~method~on~the~same~object
class~Song~{
~~~~int~seconds;
~
~~~~Song(int~s)~{
~~~~~~~~seconds~=~s;
~~~~}
~
~~~~static~int~level(int~n)~{
~~~~~~~~return~n~*~100;
~~~~}
~
~~~~boolean~discount(int~s)~{
~~~~~~~~return~s~>~level(3);
~~~~}
~
~~~~double~computePrice(double~pricePerSecond)~{
~~~~~~~~double~price~=~seconds~*~pricePerSecond;
~~~~~~~~if~(discount(seconds))
~~~~~~~~~~~~price~=~price~*~0.9;
~~~~~~~~return~price;
~~~~}
}
```

```
~
public~class~Method06B~{
~~~~public~static~void~main(/*~String[]~args~*/)~{
~~~~~~~~Song~song1~=~new~Song(164);
~~~~~~~~Song~song2~=~new~Song(403);
~~~~~~~~double~price1~=~song1.computePrice(0.01);
~~~~~~~~double~price2~=~song2.computePrice(0.01);
~~~~~~~~System.out.println(price1);
~~~~~~~~System.out.println(price2);
~~~~}
}
```

Given a call to a method `m2` within a method `m1`:

```
    void~m1()~{
~~m2();
}
```

it is impossible to tell from the call if m2 is being implicitly called *on* the same object or if it is a static method defined in the class.

- This program is a modification of the previous one: instead of comparing `s` with 300 in the method `discount`, it is compared with the value returned by the method `level`. It is impossible to tell from the calls alone to `discount` and `level` that the first is a call on an object while the second is a call to a static method.

Exercise Modify the calls to `discount` and `level` so that it is immediately apparent which is definitely a call on an object and which is definitely a call to a static method.

3.1.7 Objects as parameters

Concept A reference to an object can be an actual parameter whose corresponding formal parameter is declared to be of the same class. As with all parameters, the *value* of actual parameter is used to initialize the formal parameter, but since it is a reference that is passed, the method that is called can access fields and methods of the object. This is called *reference semantics*.

Program: Method07.java

```
    //~Learning~Object~Method07
//~~~~objects~as~parameters
class~Song~{
~~~~int~seconds;
~
~~~~Song(int~s)~{
~~~~~~~~seconds~=~s;
~~~~}
~
~~~~double~computePrice(double~pricePerSecond)~{
~~~~~~~return~seconds~*~pricePerSecond;
~~~~}
}
~
public~class~Method07~{
~
~~~~static~double~getPrice(Song~s,~double~ppS)~{
```

```
~~~~~~~~return~s.computePrice(ppS);
~~~~}
~
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~Song~song1~=~new~Song(164);
~~~~~~~~Song~song2~=~new~Song(103);
~~~~~~~~double~price1~=~getPrice(song1,~0.01);
~~~~~~~~double~price2~=~getPrice(song2,~0.02);
~~~~~~~~System.out.println(price1);
~~~~~~~~System.out.println(price2);
~~~~}
}
```

This program computes the cost of a song as the product of its length in seconds and the price per second. A class `Song` is defined to encapsulate the field `seconds` and the method `computePrice`. The method `getPrice` in the `main` method receives an object of class `Song` as a parameter and calls `computePrice`.

- Two objects of class Song are instantiated and references to them are assigned to the variables `song1` and `song2`.
- The method `getPrice` is called with two parameters: the first is a reference `song1` to an object of class `Song`, while the second is a value of type double. The actual parameters are used to initialize the formal parameters; check that `song1` and `s` reference the same object.
- Since the formal parameter `s` receives a reference to an object of class `Song` (in this case `song1`), it can be used to call the method `computePrice` declared in the class.
- The method returns a value that is assigned to `price1`.
- A second call to the method is executed exactly the same way, except that the actual parameter is the reference contained in `song2`.
- The values of `price1` and `price2` are printed.

Exercise Modify the program so that discount does not use the explicit parameter `s`.

3.1.8 Returning objects

Concept A return value can be a reference to an object.

Program: Method08.java

```
    //~Learning~Object~Method08
//~~~~returning~objects
class~Song~{
~~~~int~seconds;
~
~~~~Song(int~s)~{
~~~~~~~~seconds~=~s;
~~~~}
~
~~~~double~computePrice(double~pricePerSecond)~{
~~~~~~~return~seconds~*~pricePerSecond;
~~~~}
}
~
public~class~Method08~{
~
~~~~static~Song~longer(Song~s1,~Song~s2)~{
```

```
~~~~~~~~if~(s1.seconds~>~s2.seconds)
~~~~~~~~~~~~return~s1;
~~~~~~~~else
~~~~~~~~~~~~return~s2;
~~~~}
~
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~Song~song1~=~new~Song(164);
~~~~~~~~Song~song2~=~new~Song(103);
~~~~~~~~Song~longerSong~~=~longer(song1,~song2);
~~~~~~~~double~price2~=~longerSong.computePrice(0.01);
~~~~~~~~System.out.println(price2);
~~~~}
}
```

This program computes the cost of a song as the product of its length in seconds and the price per second. A class `Song` is defined to encapsulate the field `seconds` and the method `computePrice`. The method `longer` in the `main` method receives references to two objects of class `Song` as parameters and returns a reference to the one with the larger value of the field `seconds`.

- Two objects of class Song are instantiated and references to them are assigned to the variables `song1` and `song2`.
- The method `longer` is called with two parameters that are references to objects of class `Song`. The actual parameters are used to initialize the formal parameters; check that `song1` and `s1` reference the same object, as do `song2` and `s2`.
- Since the formal parameters `s1` and `s2` receive references to objects of class `Song`, they can be used to access the fields `seconds` of each object.
- The method returns the reference to the object whose field `seconds` has the larger value. The reference is assigned to the variable `longerSong`; check that this reference is to the same object as the reference in `song1`.
- The reference in `longerSong` is used to call the method `computePrice` and the value returned is assigned to the variable `price2`.
- The value `price2` is printed.

Exercise Modify the program so that discount does not use the explicit parameter `s`.

Exercise Replace the last declaration and statements of the program by one declaration.

Exercise Write a method to swap two integer values.

3.1.9 Returning locally instantiated objects

Concept When a method terminates, its activation record is deallocated. However, if an object has been instantiated *within the method*, a reference to the object can be returned to the calling method.

Program: Method09.java

```
    //~Learning~Object~Method09
//~~~~returning~locally~instantiated~objects
class~Song~{
~~~~int~seconds;
~
~~~~Song(int~s)~{
~~~~~~~~seconds~=~s;
~~~~}
~
```

```
~~~~double~computePrice(double~pricePerSecond)~{
~~~~~~~return~seconds~*~pricePerSecond;
~~~~}
}
~
public~class~Method09~{
~
~~~~static~Song~doubleSong(Song~s1)~{
~~~~~~~~Song~d~=~new~Song(s1.seconds*2);
~~~~~~~~return~d;
~~~~}
~
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~Song~song1~=~new~Song(164);
~~~~~~~~Song~longSong~~=~doubleSong(song1);
~~~~}
}
```

This program computes the cost of a song as the product of its length in seconds and the price per second. A class `Song` is defined to encapsulate the field `seconds` and the method `computePrice`. The method `double` in the `main` method receives a reference to an object of class `Song` as a parameter and returns a reference to an new object of class `Song` whose field `seconds` is twice as large.

- Two objects of class Song are instantiated and references to them are assigned to the variables `song1` and `song2`.
- The method `doubleSong` is called with an actual parameter that is a reference `song1` to an object of class `Song`. The actual parameter is used to initialize the formal parameter; check that `song1` and `s1` reference the same object.
- *Within the method*, the `seconds` field of the object referenced by `s1` is used to instantiate a new object whose reference is assigned to the variable `d` of class `Song`.
- The method returns the reference to the object contained in `d`; although `d` disappears when the activation record is deallocated, the object still exists as does the reference that is returned.
- The returned reference is assigned to the variable `longSong`; check that `song1` and `longSong` reference *different* objects!

Exercise Replace the last line of the program by: `song1 = doubleSong(song1);` and explain precisely what happens.

Chapter 4

Learning Objects for Arrays in Java[1]

4.1 Learning Objects for Arrays

Concept An array is a sequence of elements of the same type; the type of the elements can be a primitive types such as `int`, or a predefined or user-defined class type. To access an element of an array, an index is given; this may be any expression of type `int`, including an integer literal or a variable.

The source code of these learning objects can be found in array.zip[2].

LO	Topic	Java Files (.java)	Prerequisites
"Array objects" (Section 4.1.1: Array objects)	Array objects	Array01A, B	
"Array initializers" (Section 4.1.2: Array initializers)	Array initializers	Array02	1
"Passing arrays as parameters" (Section 4.1.3: Passing arrays as parameters)	Passing arrays as parameters	Array03	2
"Returning an array from a method" (Section 4.1.4: Returning an array from a method)	Returning an array from a method	Array04	2
			continued on next page

[1] This content is available online at <http://cnx.org/content/m31245/1.1/>.
[2] See the file at <http://cnx.org/content/m31245/latest/array.zip>

"Array assignment can create garbage" (Section 4.1.5: Array assignment can create garbage)	Array assignment can create garbage	Array05	4
"Two-dimensional arrays" (Section 4.1.6: Two-dimensional arrays)	Two-dimensional arrays	Array06	3
"Arrays of arrays" (Section 4.1.7: Arrays of arrays)	Arrays of arrays	Array07	6
"Ragged Arrays" (Section 4.1.8: Ragged Arrays)	Ragged arrays	Array08	6
"Arrays of objects" (Section 4.1.9: Arrays of objects)	Arrays of objects	Array09	3

Table 4.1

The example used in LO 1 through LO 4 is to fill an array with a sequence of fibonacci numbers (0,1,1,2,3,5,8). The programs for LO 5 through LO 8 concern matrices. The program for LO 9 is explained there.

4.1.1 Array objects

Concept An array is created in three steps: first a variable of an array type is declared; then the array is allocated; finally, the elements of the array are given values. The syntax for accessing an array `a` is `a[i]`, and the field `a.length` gives the length of the array, so that if we modify the program by changing the size of the array the rest of the program need not change.

Program: Array01A.java

```
    //~Learning~Object~Array01A
//~~~~array~objects
public~class~Array01A~{
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~int[]~fib;
~~~~~~~~fib~=~new~int[7];
~~~~~~~~fib[0]~=~0;
~~~~~~~~fib[1]~=~1;
~~~~~~~~for~(int~i~=~2;~i~<~fib.length;~i++)
~~~~~~~~~~~~fib[i]~=~fib[i-1]~+~fib[i-2];
~~~~}
}
```

The program creates an array in the three steps described above.

- Initially, the variable `fib` of type integer array (denoted `int[]`) is allocated and contains the null value.
- `new fib[7]` creates an array object with its seven fields having the default integer value zero; then the reference to the object is returned and stored in the variable `fib`.

- The length field of the array is displayed above the cells for the elements.
- A `for` loop is used to assign values to each element of the array.
- The thin white lines show the constants and expressions that are used as indices into the array.
- *Automatic dereferencing*: Although expressions like `fib[i-2]` seem to indicate that `fib` is being indexed, `fib` contains a reference to an array; an implicit operation of dereferencing is carried out to obtain the array itself from the reference and the index `[i-2]` is then applied to that array.

Concept It is possible to combine the first two steps in creating an array: declaring the array field and allocating the array object.

Program: Array01B.java

```
    //~Learning~Object~Array01B
//~~~~array~objects
public~class~Array01B~{
~~~~private~final~static~int~SIZE~=~7;
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~int[]~fib~=~new~int[SIZE];
~~~~~~~~fib[0]~=~0;
~~~~~~~~fib[1]~=~1;
~~~~~~~~for~(int~i~=~2;~i~<~fib.length;~i++)
~~~~~~~~~~~~fib[i]~=~fib[i-1]~+~fib[i-2];
~~~~}
}
```

This program combines the declaration of the array field with its allocator. We have used the constant `SIZE` to specify the size of the array; this makes it easier to modify the program; nevertheless, `fib.length` is still used in the executable statements.

- Initially, the static variable `SIZE` is created in the constant area and given its value.
- The execution of the program is as before.

Exercise Modify the program so that the fibonacci sequence appears in reverse order.

4.1.2 Array initializers

Concept An array object can be created implicitly by giving a list of values within braces.

Program: Array02.java

```
    //~Learning~Object~Array02
//~~~array~initializers
public~class~Array02~{
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~int[]~fib~=~{0,~1,~1,~2,~3,~5,~8};
~~~~}
}
```

The program initializes an array with values of the fibonacci sequence.

- Initially, the variable `fib` of type integer array (denoted `int[]`) is allocated.
- *As part of the same statement*, the array object is created and its seven fields contain the values from the initializer.
- Only then is the reference to the object returned and stored in the variable `fib`.

Exercise Can an element of an array initializer be the value of an expression containing variables previously declared? Modify this program accordingly and try to compile and run it. Explain what happens.

4.1.3 Passing arrays as parameters

Concept An array is an object. Since the array variable itself contains a reference, it can be passed as an actual paramter to a method and the reference is used to initialize the formal parameter.

Program: Array03.java

```
    //~Learning~Object~Array03
//~~~~passing~arrays~as~parameters
public~class~Array03~{
~~~~static~void~reverse(int[]~a)~{
~~~~~~~~int~temp,j;
~~~~~~~~for~(int~i~=~0;~i~<~a.length~/~2;~i++)~{
~~~~~~~~~~~~j~=~a.length-i-1;
~~~~~~~~~~~~temp~=~a[i];
~~~~~~~~~~~~a[i]~=~a[j];
~~~~~~~~~~~~a[j]~=~temp;
~~~~~~~~}
~~~~}
~
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~int[]~fib~=~{0,~1,~1,~2,~3,~5,~8};
~~~~~~~~reverse(fib);
~~~~}
}
```

This program passes an array as a parameter to a method that reverses the elements of the array.

- Initially, the variable `fib` of type integer array is allocated. As part of the same statement, the array object is created with its seven fields having the values in the initializer; the reference to the object is returned and stored in the variable `fib`.
- The array (that is, a *reference* to the array) is passed as a parameter to the method `reverse`. There are now two arrows pointing to the array: the reference from the `main` method and the reference from the parameter `a` of the method `reverse`.
- The method scans the first half of the array, exchanging each element with the corresponding one in the second half. Variables `i` and `j` contain the indices of the two elements that are exchanged.
- Upon return from the method, the variable `fib` still contains a reference to the array, which has had its sequence of values reversed.

Exercise Instead of declaring the variable `j` outside the for-loop, declare it just inside the for-loop as follows:

```
int j = a.length-i-1;
```

Trace the execution and explain what happens.

4.1.4 Returning an array from a method

Concept An array can be allocated within a method. Although the variable containing the reference to the array is local to the method, the array itself is global and the reference can be returned from the method.

Program: Array04.java

```
    //~Learning~Object~Array04
//~~~~returning~an~array~from~a~method
public~class~Array04~{
~~~~static~int[]~reverse(int[]~a)~{
```

```
~~~~~~~~int[]~b~=~new~int[a.length];
~~~~~~~~int~j;
~~~~~~~~for~(int~i~=~0;~i~<~a.length~/~2;~i++)~{
~~~~~~~~~~~~j~=~a.length-i-1;
~~~~~~~~~~~~b[j]~=~a[i];
~~~~~~~~~~~~b[i]~=~a[j];
~~~~~~~~}
~~~~~~~~return~b;
~~~~}
~
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~int[]~fib~=~{0,~1,~1,~2,~3,~5,~8};
~~~~~~~~int[]~reversedFib~=~reverse(fib);
~~~~}
}
```

This program passes an array as a parameter to a method that reverses the elements of the array. The array is reversed into a new array `b` that is allocated in the method `reverse`. It is then returned to the main method and assigned to `reversedFib`, a different variable of the same array type `int[]`.

- Initially, the variable `fib` of type integer array is allocated. As part of the same statement, the array object is created with its seven fields having the values in the initializer; the reference to the object is returned and stored in the variable `fib`.
- A reference to the array is passed as a parameter to the method `reverse`. The formal parameter `a` contains a reference to the same array pointeed to by the actual parameter `fib`.
- A new array `b` of the same type and length as the parameter `a` is declared and allocated.
- Each iteration of the for-loop moves one element from the first half of `a` to the second half of `b` and one element from the second half of `a` to the first half of `b`. Variables `i` and `j` contain the indices of the two elements that are moved.
- The reference to array `b` is returned. Although array referenced by `b` was allocated *within* the method call, it still exists after returning.
- The reference that is returned is assigned to `reversedFib`.

Exercise The program has a bug. Fix it!

4.1.5 Array assignment can create garbage

Concept Since an array variable contains a reference to the array itself, if `null` or another value (another array of the same type) is assigned to the variable, the first array may no longer be accessible. Inaccessible memory is called *garbage*. The Java runtime system includes a *garbage collector* whose task is to return garbage to the pool of memory that can be allocated.

Program: Array05.java

```
    //~Learning~Object~Array05
//~~~~array~assignment~can~create~garbage
public~class~Array05~{
~~~~static~int[]~first(int[]~a)~{
~~~~~~~~int[]~b~=~new~int[a.length/2];
~~~~~~~~for~(int~i~=~0;~i~<~a.length~/~2;~i++)
~~~~~~~~~~~~b[i]~=~a[i];
~~~~~~~~return~b;
~~~~}
~
```

```
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~int[]~fib~=~{0,~1,~1,~2,~3,~5,~8};
~~~~~~~~fib~=~first(fib);
~~~~}
}
```

An array referenced by the variable `fib` is passed as a parameter to a method that moves the values of the elements in the first half of the array `fib` into a new array `b` which is allocated in the method. The new array is returned to the main method and assigned to the variable `fib`, destroying the reference to the original array.

- Initially, the variable `fib` of type integer array is allocated. As part of the same statement, the array object is created with its seven fields having the values in the initializer; the reference to the object is returned and stored in the variable `fib`.
- A reference to the array is passed as a parameter to the method `first`. The formal parameter `a` contains a reference to the same array pointed to by the actual parameter `fib`.
- A new array `b` of the same type as the parameter `a` but half the length is declared and allocated.
- Each iteration of the for-loop moves one element from the first half of `a` to the corresponding element in the array `b`.
- The reference to array `b` is returned. Although array referenced by `b` was allocated *within* the method call, it still exists after returning.
- There are no references to the original array so it is inaccessible. Jeliot does not visualize garbage collection so the array remains visualized in the Instance and Array Area until the end of the program.

Exercise Modify the program so that the original array remains accessible in a different field.

4.1.6 Two-dimensional arrays

Concept A matrix can be stored in a two-dimensional array. The syntax is `int[][]` with two indices, the first for rows and the second for columns. To access an element of the array, expressions for the two indices must be givien.

Program: Array06.java

```
//~Learning~Object~Array06
//~~~~~two-dimensional~arrays
public~class~Array06~{
~~~~static~int~addElements(int[][]~a)~{
~~~~~~~~int~sum~=~0;
~~~~~~~~for~(int~i~=~0;~i~<~a.length;~i++)
~~~~~~~~~~~~for~(int~j~=~0;~j~<~a[i].length;~j++)
~~~~~~~~~~~~~~~~sum~=~sum~+~a[i][j];
~~~~~~~~return~sum;
~~~~}
~
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~int[][]~matrix~=~new~int[2][2];
~~~~~~~~for~(int~i~=~0;~i~<~matrix.length;~i++)
~~~~~~~~~~~~for~(int~j~=~0;~j~<~matrix[i].length;~j++)
~~~~~~~~~~~~~~~~matrix[i][j]~=~i*matrix.length~+~j;
~~~~~~~~int~sum~=~addElements(matrix);
~~~~}
}
```

This program creates a 2×2 matrix and computes the sum of its elements.

- A two-dimensional array is allocated, the reference to it is assigned to the variable `matrix`. The variable `matrix` contains one reference for each row, and the rows are allocated as separate objects. Note that Jeliot displays each row from top to bottom as it does for all objects!
- The elements of the array are initialized to (0,1,2,3) in a nested for-loop. The outside loop iterates over the rows and the inner loop iterates over the columns within an array.
- `matrix.length` is used to get the number of rows and `matrix[i].length` to get the number of columns in row `i`, which is the same for all rows in this program.
- The reference to the array is passed as a parameter to the method `addElements`, which adds the values of all the elements.
- The sum is returned from the method and assigned to the variable `sum`.

Exercise Modify the program perform the same computation on a 2×3 matrix and on a 3×2 matrix.

4.1.7 Arrays of arrays

Concept A two-dimensional array is really an array of arrays; that is, each element of the array contains a reference to another array. Therefore, by using only one index a one-dimensional array is obtained.

Program: Array07.java

```
    //~Learning~Object~Array07
//~~~~arrays~of~arrays
public~class~Array07~{
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~int[][]~matrix~=~new~int[2][2];
~~~~~~~~for~(int~i~=~0;~i~<~matrix.length;~i++)
~~~~~~~~~~~~for~(int~j~=~0;~j~<~matrix[i].length;~j++)
~~~~~~~~~~~~~~~~matrix[i][j]~=~i*matrix.length~+~j;
~~~~~~~~int[]~vector~=~matrix[1];
~~~~}
}
```

This program creates a 2×2 matrix and then assigns the second row to a variable of type *one-dimensional* array.

- A two-dimensional array is allocated, the reference to it is assigned to the variable `matrix`. The variable `matrix` contains one reference for each row and the rows are allocated as separate objects. Note that Jeliot displays rows from top to bottom as it does for all objects!
- The elements of the array are initialized to (0,1,2,3) in a nested for-loop. The outside loop iterates over the rows and the inner loop iterates over the columns within an array.
- `matrix.length` is used to get the number of rows and `matrix[i].length` to get the number of columns in row `i`, which is the same for all rows in this program.
- A variable `vector` of type one-dimensional array is declared and initialized with the second row of the matrix, `matrix[1]`.

Exercise Write a program to rotate the rows of the array `matrix`. That is, row 0 becomes row 1 and row 1 becomes row 0. Now do this for an array of size 3×3: row 0 becomes row 1, row 1 becomes row 2 and row 2 becomes row 0.

4.1.8 Ragged Arrays

Concept A two-dimensional array is really an array of arrays; that is, each element of the array contains a reference to another array. However, the two-dimensional array need not be a square matrix, and each row can have a different number of elements. By using only one index a one-dimensional array is obtained and these arrays need not all be of the same size.

Program: Array08.java

```
//~Learning~Object~Array08
//~~~~ragged~arrays
public~class~Array08~{
~~~~static~int~addElements(int[][]~a)~{
~~~~~~~~int~sum~=~0;
~~~~~~~~for~(int~i~=~0;~i~<~a.length;~i++)
~~~~~~~~~~~~for~(int~j~=~0;~j~<~a[i].length;~j++)
~~~~~~~~~~~~~~~~sum~=~sum~+~a[i][j];
~~~~~~~~return~sum;
~~~~}
~
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~int[][]~matrix~=~new~int[3][];
~~~~~~~~int[]~row0~=~{0,~1,~2};
~~~~~~~~int[]~row1~=~{3,~4};
~~~~~~~~int[]~row2~=~{5};
~~~~~~~~matrix[0]~=~row0;
~~~~~~~~matrix[1]~=~row1;
~~~~~~~~matrix[2]~=~row2;
~~~~~~~~int~sum~=~addElements(matrix);
~~~~}
}
```

Here we create the upper-left triangle of a 3×3 matrix: row 0 of length 3, row 1 of length 2 and row 2 of length 1. Then we add the elements of the "ragged" array.

- The variable `matrix` is allocated, but since the size of the rows is not given, it is allocated as a one-dimensional array whose elements are references to one-dimensional arrays of integers. The default value for the elements is `null`.
- Three rows of different size are allocated with initializers and assigned to the elements of the array `matrix`.
- A reference to `matrix` is passed to the method `addElements` which adds the elements of the array and returns the value.
- `matrix.length` is used to get the number of rows and `matrix[i].length` to get the number of columns in row `i`; these are different for each row.

Exercise Simplify the allocation of the array `matrix`. First, show how the variables `row` can be eliminated. Then find out how to write an initializer for a two-dimensional array so that the array can be initialized in one declaration. (Note: initializers for two-dimensional arrays are not supported in Jeliot.)

4.1.9 Arrays of objects

Concept Arrays can contain references to arbitrary objects. There is no difference between these arrays and arrays whose values are of primitive type, except that an individual element can be of any type.

Program: Array09.java

```
//~Learning~Object~Array09
//~~~~arrays~of~objects
class~Access~{
~~~~String~name;
~~~~int~level;
```

```
~~~~Access(String~u,~int~l)~{
~~~~~~~~name~=~u;~level~=~l;
~~~~}
}
~
public~class~Array09~{
~~~~static~void~swap(Access[]~a,~int~i,~int~j)~{
~~~~~~~~Access~temp~=~a[i];
~~~~~~~~a[i]~=~a[j];
~~~~~~~~a[j]~=~temp;
~~~~}
~
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~Access[]~accesses~=~new~Access[2];
~~~~~~~~accesses[0]~=~new~Access("Bob",~3);
~~~~~~~~accesses[1]~=~new~Access("Alice",~4);
~~~~~~~~swap(accesses,~0,~1);
~~~~}
}
```

Objects of class `Access` contain name of a bank customer and the access level permitted for that customer. This program creates two objects, assigns the their references to elements of the `Access` and then swaps the elements of the array.

- The array `accessess` of type `Access[]` is allocated and contains null references.
- An object of type `Access` are allocated and initialized by its constructor; a reference to the object is stored in the first element of the array `accessess`.
- Similarly, another object is created and stored in the second element.
- The array `accessess` is passed to the method `swap` along with the indices 0 and 1.
- The two elements of the array are swapped. Note that after executing `a[i] = a[j]`, both elements of the array point to the second object (`Alice`,4), while a reference to the first object (`Bob`,3) is saved in the variable `temp`.

Exercise Modify the program so that the initialization of the array `accessess` is done in one statement instead of three.

Exercise Explain what happens if the method `swap` is replaced by:

```
    static~void~swap(Access~a,~Access~b)~{
~~Access~temp~=~a;
~~a~=~b;
~~b~=~temp;
}
```

and the call by `swap(accesses[0], accesses[1]);`

Chapter 5

Learning Objects for Constructors in Java[1]

5.1 Learning Objects for Constructors

Concept The process of creating an object involves allocating memory for the object and assigning the reference to this block of memory to a variable. *Constructors* enable arbitrary initialization of the object during its creation.

These source code of these learning objects can be found in constructor.zip[2].

LO	Topic	Java Files (.java)	Prerequisites
"What are constructors for?" (Section 5.1.1: What are constructors for?)	What are constructors for?	Constructor01A, B, C	
"Computation within constructors" (Section 5.1.2: Computation within constructors)	Computation within constructors	Constructor02	1
"Overloading constructors" (Section 5.1.3: Overloading constructors)	Overloading constructors	Constructor03	2
continued on next page			

[1]This content is available online at <http://cnx.org/content/m31248/1.1/>.
[2]See the file at <http://cnx.org/content/m31248/latest/constructor.zip>

"Invoking an overloaded constructor from within a constructor" (Section 5.1.4: Invoking an overloaded constructor from within a constructor)	Invoking another constructor	Constructor04	3
"Explicit default constructors" (Section 5.1.5: Explicit default constructors)	Explicit default constructors	Constructor05	3
"Constructors for subclasses" (Section 5.1.6: Constructors for subclasses)	Constructors for subclasses	Constructor06A, B, C	3
"Constructors with object parameters" (Section 5.1.7: Constructors with object parameters)	Constructors with object parameters	Constructor07	3
"Constructors with subclass object parameters" (Section 5.1.8: Constructors with subclass object parameters)	Constructors with subclass		
	object parameters	Constructor08	6, 7

Table 5.1

Program The example used in these LOs is class `Song` with three fields: the `name` of the song, the length of the song in `seconds` and the `pricePerSecond`. The class is to be used to implement a website which charges for downloading the song; the price is the product of the length of the song in second and the price per second. To focus the discussion on constructors, the fields are not declared private.

5.1.1 What are constructors for?

Concept An object is created by allocating memory for its fields. The fields are given the default values for their types. A reference to the object is returned and assigned to a variable; the reference can be used to access the fields and methods of the object.

Program: Constructor01B.java

```
    //~Learning~Object~Constructor01A
//~~~~what~are~constructors~for?
class~Song~{
~~~~String~name;
~~~~int~seconds;
~~~~double~pricePerSecond;
~
~~~~public~double~computePrice()~{
~~~~~~~~return~seconds~*~pricePerSecond;
~~~~}
```

```
}
~
public~class~Constructor01A~{
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~Song~song1~=~new~Song();
~~~~~~~~song1.name~=~"Waterloo";
~~~~~~~~song1.seconds~=~164;
~~~~~~~~song1.pricePerSecond~=~0.01;
~~~~~~~~double~price~=~song1.computePrice();
~~~~}
}
```

If a constructor is not explicitly declared a default constructor is called.

- The variable `song1` is allocated and contains the null value.
- Memory is allocated for the fields of the object; this is displayed in the Instance and Array Area. Default values are assigned to the three fields.
- The default constructor is called but does nothing except return a reference to the object.
- The reference is stored in the variable `song1`.
- The reference in `song1` is used to assign values to the fields of the object.
- The reference in `song1` is used to call the method `computePrice` on the object; the method computes and returns the price, which is assigned to the variable `price`.

Concept An explicit constructor method can be declared and used to initialize each object. The constructor method is identified by a special syntax: the name of the method is the same as the name of the class and *there is no return type* (because the value returned is of the type of the class itself).

Program: Constructor01B.java

```
    //~Learning~Object~Constructor01B
//~~~~what~are~constructors~for?
class~Song~{
~~~~String~name;
~~~~int~seconds;
~~~~double~pricePerSecond;
~
~~~~Song()~{
~~~~~~~~name~=~"Waterloo";
~~~~~~~~seconds~=~164;
~~~~~~~~pricePerSecond~=~0.01;
~~~~}
~
~~~~public~double~computePrice()~{
~~~~~~~~return~seconds~*~pricePerSecond;
~~~~}
}
~
public~class~Constructor01B~{
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~Song~song1~=~new~Song();
~~~~~~~~double~price~=~song1.computePrice();
~~~~}
}
```

This program is the same as the previous one except that the assignment of nondefault values to the fields of the object is moved to an explicit constructor.

- The variable `song1` is allocated and contains the null value.
- Memory is allocated for the fields of the object; this is displayed in the Instance and Array Area. Default values are assigned to the three fields.
- The constructor is called and assigns values to the three fields; then it returns a reference to the object.
- The reference is stored in the variable `song1`.
- The reference in `song1` is used to call the method `computePrice` on the object; the method computes and returns the price, which is assigned to the variable `price`.

Exercise Add the creation of a second object `song2` to the program and verify that it is initialized to the same values.

Concept Of course, it is highly unlikely that all objects created from a class will be initialized with the same values. A constructor can have formal parameters like any other method and is called with actual parameters.

Program: Constructor01C.java

```
    //~Learning~Object~Constructor01C
//~~~~what~are~constructors~for?
class~Song~{
~~~~String~name;
~~~~int~seconds;
~~~~double~pricePerSecond;
~
~~~~Song(String~n,~int~s,~double~p)~{
~~~~~~~~name~=~n;
~~~~~~~~seconds~=~s;
~~~~~~~~pricePerSecond~=~p;
~~~~}
~
~~~~public~double~computePrice()~{
~~~~~~~~return~seconds~*~pricePerSecond;
~~~~}
}
~
public~class~Constructor01C~{
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~Song~song1~=~new~Song("Waterloo",~164,~0.01);
~~~~~~~~double~price~=~song1.computePrice();
~~~~}
}
```

This program is the same as the previous one except that the constructor has formal parameters and the actual parameters passed to the constructor are assigned to the fields of the object.

- The variable `song1` is allocated and contains the null value.
- Memory is allocated for the fields of the object; this is displayed in the Instance and Array Area. Default values are assigned to the three fields.
- The constructor is called with three actual parameters; these values are assigned to the formal parameters of the constructor method.
- The values of the formal parameters are assigned to the three fields; then the constructor returns a reference to the object.

- The reference is stored in the variable `song1`.
- The reference in `song1` is used to call the method `computePrice` on the object; the method computes and returns the price, which is assigned to the variable `price`.

Exercise Modify the class so that the second parameter passes the number of minutes; the value of the field `seconds` will have to be computed in the constructor.

5.1.2 Computation within constructors

Concept Constructors are often used simply for assigning initial values to fields of an object; however, an arbitrary initializing computation can be carried out within the constructor.

Program: Constructor02.java

```
//~Learning~Object~Constructor02
//~~~~computation~within~constructors
class~Song~{
~~~~String~name;
~~~~int~seconds;
~~~~double~pricePerSecond;
~~~~double~price;
~
~~~~Song(String~n,~int~s,~double~p)~{
~~~~~~~~name~=~n;
~~~~~~~~seconds~=~s;
~~~~~~~~pricePerSecond~=~p;
~~~~~~~~price~=~computePrice();
~~~~}
~
~~~~private~double~computePrice()~{
~~~~~~~~return~seconds~*~pricePerSecond;
~~~~}
}
~
public~class~Constructor02~{
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~Song~song1~=~new~Song("Waterloo",~164,~0.01);
~~~~}
}
```

The price of a song will not change as long as the fields `second` and `pricePerSecond` do not change; to avoid recomputing the price each time it is needed, the class contains a field `price` whose value is computed *within* the constructor. The method `computePrice` is declared to be `private` because it is needed only by the constructor.

- The variable `song1` is allocated and contains the null value.
- Memory is allocated for the fields of the object and default values are assigned to the three fields.
- The constructor is called with three actual parameters; these values are assigned to the formal parameters of the constructor method and the values of the formal parameters are assigned to the three fields.

- The method `computePrice` is called; it returns a value which stored in the field `price`.
- The constructor returns a reference to the object, which is stored in the variable `song1`. The field `price` can be accessed to obtain the price of a song.

Exercise Modify the class so that no song has a price greater than two currency units.

5.1.3 Overloading constructors

Concept Constructors can be *overloaded* like other methods. A method is overloaded when there is more than one method with the same name; the parameter signature is used to decide which method to call. For constructors, overloading is usually done when some of the fields of an object can be initialized with default values, although we want to retain the possibility of explicitly supplying all the initial values.

Program: Constructor03.java

```
//~Learning~Object~Constructor03
//~~~~overloading~constructors
class~Song~{
~~~~String~name;
~~~~int~seconds;
~~~~double~pricePerSecond;
~~~~double~price;
~~~~final~static~double~DEFAULT_PRICE~=~0.005;
~
~~~~Song(String~n,~int~s,~double~p)~{
~~~~~~~~name~=~n;
~~~~~~~~seconds~=~s;
~~~~~~~~pricePerSecond~=~p;
~~~~~~~~price~=~computePrice();
~~~~}
~
~~~~Song(String~n,~int~s)~{
~~~~~~~~name~=~n;
~~~~~~~~seconds~=~s;
~~~~~~~~pricePerSecond~=~DEFAULT_PRICE;
~~~~~~~~price~=~computePrice();
~~~~}
~
~~~~private~double~computePrice()~{
~~~~~~~~return~seconds~*~pricePerSecond;
~~~~}
}
~
public~class~Constructor03~{
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~Song~song1~=~new~Song("Waterloo",~164,~0.01);
~~~~~~~~Song~song2~=~new~Song("Fernando",~253);
~~~~}
}
```

The website charges a uniform price per second for all songs, except for special offers. We define two constructors, one that specifies a price for special offers and another that uses a default price for ordinary songs.

- The value of the static constant `DEFAULT_PRICE` is set as soon as the class is loaded and is displayed in the Constant area.
- The variable `song1` is allocated and contains the null value.
- Memory is allocated for the *four* fields of the object and default values are assigned to the fields.
- The constructor is called with *three* actual parameters; the call is resolved so that the first constructor is executed. These values are assigned to the formal parameters of the constructor method and the values of the formal parameters are assigned to the three fields.

- The method `computePrice` is called; it returns a value which stored in the field `price`.
- The constructor returns a reference to the object, which is stored in the variable `song1`.
- The computation is then repeated for `song2`. Since the constructor is called with just two parameters (for `name` and `seconds`), the second constructor is executed. The value of the field `pricePerSecond` is assigned from the constant, not from a parameter.

Exercise Modify the class to include a constructor with one parameter for the `name` and with a default song length of three minutes.

Exercise Modify the class to include a constructor with no parameters, so that all fields receive default values. Is there any meaning to the following constructor?

```
    Song()~{
}
```

5.1.4 Invoking an overloaded constructor from within a constructor

Concept Constructors can be *overloaded* like other methods. A method is overloaded when there is more than one method with the same name; the parameter signature is used to decide which method to call. For constructors, overloading is usually done when some of the fields of an object can be initialized with default values, although we want to retain the possibility of explicitly supplying all the initial values. In such cases, it is convenient to invoke one constructor from within another in order to avoid duplicating code. Invoking the method `this` within one constructor calls another constructor with the appropriate parameter signature.

Program: Constructor04.java

```
    //~Learning~Object~Constructor04
//~~~~invoking~one~constructor~from~another
class~Song~{
~~~~String~name;
~~~~int~seconds;
~~~~double~pricePerSecond;
~~~~double~price;
~~~~final~static~double~DEFAULT_PRICE~=~0.005;
~
~~~~Song(String~n,~int~s,~double~p)~{
~~~~~~~~name~=~n;
~~~~~~~~seconds~=~s;
~~~~~~~~pricePerSecond~=~p;
~~~~~~~~price~=~computePrice();
~~~~}
~
~~~~Song(String~n,~int~s)~{
~~~~~~~~this(n,~s,~DEFAULT_PRICE);
~~~~}
~
~~~~private~double~computePrice()~{
~~~~~~~~return~seconds~*~pricePerSecond;
~~~~}
}
~
public~class~Constructor04~{
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~Song~song1~=~new~Song("Waterloo",~164);
```

```
~~~~}
}
```

The website charges a uniform price per second for all songs, except for special offers. We define two constructors, one that specifies a price for special offers and another that uses a default price for ordinary songs.

- The value of the static constant `DEFAULT_PRICE` is set as soon as the class is loaded and is displayed in the Constant area.
- The variable `song1` is allocated and contains the null value.
- Memory is allocated for the *four* fields of the object and default values are assigned to the fields.
- The constructor is called with *two* actual parameters; the call is resolved so that it is the second constructor that is executed.
- The two parameters, together with the default price, are immediately used to call the first constructor that has three parameters. The method name `this` means: call a constructor from *this* class. This constructor initializes the first three fields from the parameters, and the value of the fourth field is computed by calling the method `computePrice`.
- The constructor returns a reference to the object, which is stored in the variable `song1`.

Exercise Modify the class to include a constructor with one parameter, the name, and with a default song length of three minutes. Can this constructor call the two-parameter constructor which in turn calls the three-parameter constructor? Can a constructor call *two* other constructors, one after another?

5.1.5 Explicit default constructors

Concept When no constructor is explicitly written in a class, a default implicit constructor with no parameters exists; this constructor does nothing. If, however, one or more explicit constructors are given, there is no longer a constructor with no parameters. Should you want one, you have to write it explicitly.

Program: Constructor05.java

```
    //~Learning~Object~Constructor05
//~~~~explicit~default~constructors
class~Song~{
~~~~String~name;
~~~~int~seconds;
~~~~double~pricePerSecond;
~~~~double~price;
~
~~~~Song(String~n,~int~s,~double~p)~{
~~~~~~~~name~=~n;
~~~~~~~~seconds~=~s;
~~~~~~~~pricePerSecond~=~p;
~~~~~~~~price~=~computePrice();
~~~~}
~
~~~~Song()~{
~~~~~~~~this("No~song",~0,~0.0);
~~~~}
~
~~~~private~double~computePrice()~{
~~~~~~~~return~seconds~*~pricePerSecond;
~~~~}
}
```

```
~
public~class~Constructor05~{
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~Song~song1~=~new~Song();
~~~~}
}
```

This program includes an explicit constructor with no parameters that calls the constructor with three parameters to perform initialization.

- The variable `song1` is allocated and contains the null value.
- Memory is allocated for the *four* fields of the object and default values are assigned to the fields.
- The constructor is called with *no* actual parameters; the call is resolved so that it is the second constructor that is executed.
- Three constant values are used to call the first constructor. The method name `this` means: call a constructor from *this* class. This constructor initializes the first three fields from the parameters, and the value of the fourth field is computed by calling the method `computePrice`.
- The constructor returns a reference to the object, which is stored in the variable `song1`.

Exercise Modify the class so that the constructor without parameters obtains initial values from the input.

5.1.6 Constructors for subclasses

Concept Constructors are *not* inherited. You must explicitly define a constructor for a subclass (with or without parameters). As its first statement, the constructor for the subclass must call a constructor for the superclass using the method `super`.

Program: Constructor06A.java

```
    //~Learning~Object~Constructor06A
//~~~~constructors~for~subclasses
class~Song~{
~~~~String~name;
~~~~int~seconds;
~~~~double~pricePerSecond;
~~~~double~price;
~
~~~~Song(String~n,~int~s,~double~p)~{
~~~~~~~~name~=~n;
~~~~~~~~seconds~=~s;
~~~~~~~~pricePerSecond~=~p;
~~~~~~~~price~=~computePrice();
~~~~}
~
~~~~private~double~computePrice()~{
~~~~~~~~return~seconds~*~pricePerSecond;
~~~~}
}
~
class~DiscountSong~extends~Song~{
~~~~double~discount;
~
~~~~DiscountSong(String~n,~int~s,~double~p,~double~d)~{
~~~~~~~~super(n,~s,~p);
```

```
~~~~~~~~discount~=~d;
~~~~}
~
~~~~private~double~computePrice()~{
~~~~~~~~return~seconds~*~pricePerSecond~*~discount;
~~~~}
}
~
public~class~Constructor06A~{
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~DiscountSong~song1~=~new~DiscountSong("Waterloo",~164,~0.01,~0.8);
~~~~~~~~double~price~=~song1.price;
~~~~}
}
```

The website wants to sell certain songs at a discount. The subclass `DiscountSong` inherits from class `Song`, adds a field `discount` and overrides `computePrice` to include `discount` in the computation. The constructor for the subclass calls the three-parameter constructor for the superclass, passing it the three parameters that it expects. The fourth parameter is used directly in the constructor `DiscountSong` to initialize the field `discount`.

- The variable `song1` is allocated and contains the null value.
- Memory is allocated for the *five* fields of the object of the subclass `DiscountSong` and default values are assigned to the fields. Four fields inherited from the superclass and one field `discount` added by the subclass.
- The constructor for the subclass `DiscountSong` is called with four parameters. It calls the constructor for the superclass `Song` which assigns values to three fields from the parameters and the fourth by calling `computePrice`.
- The superclass constructor returns and then the fourth parameter of the subclass constructor is assigned to the field `discount`.
- The reference to the subclass object is returned and assigned to a variable `song1` of that type.

Unfortunately, this does not do what we intended, because the superclass method for `computePrice` is used to compute `price` instead of the method from the subclass.

Exercise Could `song1` be declared to be of type `Song`? Explain your answer.

Program: Constructor06B.java

```
  //~Learning~Object~Constructor06B
//~~~~constructors~for~subclasses
class~Song~{
~~~~String~name;
~~~~int~seconds;
~~~~double~pricePerSecond;
~~~~double~price;
~
~~~~Song(String~n,~int~s,~double~p)~{
~~~~~~~~name~=~n;
~~~~~~~~seconds~=~s;
~~~~~~~~pricePerSecond~=~p;
~~~~~~~~price~=~computePrice();
~~~~}
~
~~~~private~double~computePrice()~{
```

```
~~~~~~~~return~seconds~*~pricePerSecond;
~~~~}
}
~
class~DiscountSong~extends~Song~{
~~~~double~discount;
~
~~~~DiscountSong(String~n,~int~s,~double~p,~double~d)~{
~~~~~~~~super(n,~s,~p);
~~~~~~~~discount~=~d;
~~~~~~~~price~=~computePrice();
~~~~}
~
~~~~private~double~computePrice()~{
~~~~~~~~return~seconds~*~pricePerSecond~*~discount;
~~~~}
}
~
public~class~Constructor06B~{
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~DiscountSong~song1~=~new~DiscountSong("Waterloo",~164,~0.01,~0.8);
~~~~~~~~double~price~=~song1.price;
~~~~}
}
```

The problem can be solved by adding a call to `computePrice` in the constructor for the subclass.

Check this by executing the code and ensuring that the discounted price is computed.

The disadvantage of this solution is that we are calling `computePrice` twice.

Program: Constructor06C.java

```
    //~Learning~Object~Constructor06C
//~~~~constructors~for~subclasses
class~Song~{
~~~~String~name;
~~~~int~seconds;
~~~~double~pricePerSecond;
~
~~~~Song(String~n,~int~s,~double~p)~{
~~~~~~~~name~=~n;
~~~~~~~~seconds~=~s;
~~~~~~~~pricePerSecond~=~p;
~~~~}
~
~~~~public~double~getPrice()~{
~~~~~~~~return~seconds~*~pricePerSecond;
~~~~}
}
~
class~DiscountSong~extends~Song~{
~~~~double~discount;
~
~~~~DiscountSong(String~n,~int~s,~double~p,~double~d)~{
```

```
~~~~~~~~super(n,~s,~p);
~~~~~~~~discount~=~d;
~~~~}
~
~~~~public~double~getPrice()~{
~~~~~~~~return~seconds~*~pricePerSecond~*~discount;
~~~~}
}
~
public~class~Constructor06C~{
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~Song~song1~=~new~DiscountSong("Waterloo",~164,~0.01,~0.8);
~~~~~~~~double~price~=~song1.getPrice();
~~~~}
}
```

Normally in an object-oriented program, all the fields of an object are private and an accessor method like `getPrice()` is used to access the values of the fields. If this is done, the computation of the price can be placed in the accessor for the superclass and overridden in accessor for the subclass.

Check this by executing the code and ensuring that the discounted price is computed.

The disadvantage of this solution is that the computation is performed for each access of the field `price`.

Exercise Develop other solutions for this problem: (a) Call `computePrice` explicitly after the call to the constructor; (b) Modify `getPrice` to compute the value of `price` on the first call and save it for future calls. Summarize the advantages and disadvantages of all the solutions for this problem.

5.1.7 Constructors with object parameters

Concept An object can contain fields of other user-defined objects, not just of primitive and predefined types. There is no difference in the constructors, except that references to objects are passed as actual parameters and assigned to fields of the object.

Program: Constructor07.java

```
    //~Learning~Object~Constructor07
//~~~~constructors~with~object~parameters
class~Song~{
~~~~String~name;
~~~~int~seconds;
~~~~double~pricePerSecond;
~
~~~~Song(String~n,~int~s,~double~p)~{
~~~~~~~~name~=~n;
~~~~~~~~seconds~=~s;
~~~~~~~~pricePerSecond~=~p;
~~~~}
~
~~~~public~double~computePrice()~{
~~~~~~~~return~seconds~*~pricePerSecond;
~~~~}
}
~
class~SongSet~{
~~~~public~Song~track1,~track2;
```

```
~
~~~~public~SongSet(Song~t1,~Song~t2)~{
~~~~~~~~track1~=~t1;~track2~=~t2;
~~~~}
}
~
public~class~Constructor07~{
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~Song~song1~=~new~Song("Waterloo",~164,~0.01);
~~~~~~~~Song~song2~=~new~Song("Fernando",~253,~0.01);
~~~~~~~~SongSet~set~=~new~SongSet(song1,~song2);
~~~~~~~~double~price1~=~set.track1.computePrice();
~~~~~~~~double~price2~=~set.track2.computePrice();
~~~~}
}
```

Two objects of type `Song` are allocated and assigned to fields of another object of type `SongSet` which has two fields of type `Song`.

- Execute the program until the two objects of type `Song` are allocated and their references assigned to the variables `song1` and `song2`. (You may want to select `Animation / Run Until (ctrl-T)` to skip the animation of these declarations.)
- A variable `set` is allocated. An object of type `SongSet` is allocated with default null fields.
- The constructor for `SongSet` is called and the references in the two variables `song1` and `song2` are passed as actual parameters. These references are stored in the two fields `track1` and `track2`.
- The reference to the object of class `SongSet` is returned and stored in `set`.
- The prices of the two objects are obtained and stored in the variables `price1` and `price2`. `set` is an object of type `Songset`, while `set.track1` is an object of type `Song` and thus can be used to call the method `computePrice`.

Exercise Modify the program so that the variables `song1` and `song2` are not used; instead, the constructors for the songs are embedded within the constructor call for `SongSet`.

Exercise Modify the program so the constructors for the songs are call within the constructor for `SongSet`. Under what circumstances would this be done?

5.1.8 Constructors with subclass object parameters

Concept An object of a subclass is also an object of the type of the superclass. Therefore, it can be used when an actual parameter is expected.

Program: Constructor08.java

```
    //~Learning~Object~Constructor08
//~~~~constructors~with~subclass~object~parameters
class~Song~{
~~~~String~name;
~~~~int~seconds;
~~~~double~pricePerSecond;
~
~~~~Song(String~n,~int~s,~double~p)~{
~~~~~~~~name~=~n;
~~~~~~~~seconds~=~s;
~~~~~~~~pricePerSecond~=~p;
~~~~}
```

```
~
~~~~public~double~computePrice()~{
~~~~~~~~return~seconds~*~pricePerSecond;
~~~~}
}
~
class~DiscountSong~extends~Song~{
~~~~double~discount;
~
~~~~DiscountSong(String~n,~int~s,~double~p,~double~d)~{
~~~~~~~~super(n,~s,~p);
~~~~~~~~discount~=~d;
~~~~}
~
~~~~public~double~computePrice()~{
~~~~~~~~return~seconds~*~pricePerSecond~*~discount;
~~~~}
}
~
class~SongSet~{
~~~~public~Song~track1,~track2;
~
~~~~public~SongSet(Song~t1,~Song~t2)~{
~~~~~~~~track1~=~t1;~track2~=~t2;
~~~~}
}
~
public~class~Constructor08~{
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~Song~song1~=~new~Song("Waterloo",~164,~0.01);
~~~~~~~~DiscountSong~song2~=~new~DiscountSong("Fernando",~253,~0.01,~0.8);
~~~~~~~~SongSet~set~=~new~SongSet(song1,~song2);
~~~~~~~~double~price1~=~set.track1.computePrice();
~~~~~~~~double~price2~=~set.track2.computePrice();
~~~~}
}
```

We allocate two objects, one of type `Song` and one of type `DiscountSong`, and use them as actual parameters in the constructor for an object of type `SongSet` that expects two parameters of type `Song`.

- Execute the program until the two objects one of type `Song` the other of type `DiscountSong` are allocated and their references assigned to the variables `song1` and `song2`, respectively. (You may want to select `Animation / Run Until (ctrl-T)` to skip the animation of these declarations.)
- The variable `set` is allocated, and an object of type `SongSet` is allocated with default null fields.
- The constructor for `SongSet` is called and the references in the two variables `song1` and `song2` are passed as actual parameters. These references are stored in the two fields `track1` and `track2`.
- The reference to the object of class `SongSet` is returned and stored in `set`.
- The prices of the two objects are obtained and stored in the variables `price1` and `price2`. `set` is an object of type `Songset`, while `set.track1` is an object of type `Song` and thus can be used to call the method `computePrice` of that class. Similarly for `price2`, except that `set.track2` is an object of type `DiscountSong`; check that the method `computePrice` of this class is called.

Exercise Can `s2` in the main method be declared to be of type `Song`? Explain.

Chapter 6

Learning Objects for Inheritance in Java[1]

6.1 Learning Objects for Inheritance

Concept *Inheritance* is an important technique for structuring object-oriented programs. Given a class (called a *superclass*) it can be extended to a *subclass*. The subclass inherits all the fields of the superclass and it can add additional fields. The subclass inherits methods of the superclass and it can add new methods or override the inherited methods with its own versions.

These source code of these learning objects can be found in inheritance.zip[2].

LO	Topic	Java Files (.java)	Prerequisites
"Inheriting fields" (Section 6.1.1: Inheriting fields)	Inheriting fields	Inheritance01	
"Inheriting methods" (Section 6.1.2: Inheriting methods)	Inheriting and overriding methods	Inheritance02	1
"Dynamic dispatching" (Section 6.1.3: Dynamic dispatching)	Dynamic dispatching	Inheritance03	2
"Downcasting" (Section 6.1.4: Downcasting)	Downcasting	Inheritance04	3
"Heterogeneous data structures" (Section 6.1.5: Heterogeneous data structures)	Heterogeneous data structures	Inheritance05	4
		continued on next page	

[1] This content is available online at <http://cnx.org/content/m31249/1.1/>.
[2] See the file at <http://cnx.org/content/m31249/latest/inheritance.zip>

"Abstract classes" (Section 6.1.6: Abstract classes)	Abstract classes	Inheritance06	5
"Equals" (Section 6.1.7: Equals)	Equals	Inheritance07A, B, C	2
"Clone" (Section 6.1.8: Clone)	Clone	Inheritance08	2
"Overloading vs. overriding" (Section 6.1.9: Overloading vs. overriding)	Overloading vs. overriding	Inheritance09	3

Table 6.1

Program The running example is a framework for the simulation of moving particles. There is a class `Particle` with a field `position` that is updated by the method `newPosition`. There are three subclasses:

- `AParticle` is derived directly from `Particle` and adds the field `spin`. The method `newPosition` is overridden.
- `BParticle` is derived directly from `Particle` and adds the field `charge`. The method `newPosition` is *not* overridden.
- `CParticle` is derived directly from `BParticle` and thus indirectly from `AParticle`. It adds the field `strange`; the method `newPosition` is overridden.

For each program the following initialization is performed and will not be explicitly mentioned for each learning object; instead, the step "the objects are created" will be listed:

- Variable are declared and assigned the null value.
- Memory is allocated for each object's fields and are given default values. For a subclass, these fields include all fields of its superclasses.
- In the constructors, a subclass calls `super` to initialize the fields declared by the superclasses and then initializes its own fields.

Tip: Use `Animation / Run Until ...` to skip over the animation of the initialization.

Tip: Several of the LOs will ask you to check that a certain version of a method is called. This can be done by looking at the source code in the left panel: the method called is highlighted in blue.

6.1.1 Inheriting fields

Concept Subclasses inherit all the fields of its superclasses; they can also add fields of their own.

Program: Inheritance01.java

```
//~Learning~Object~Inheritance01
//~~~~inheriting~fields
class~Particle~{
~~~~int~position;
~
~~~~Particle(int~p)~{
~~~~~~~~position~=~p;
~~~~}
~
~~~~void~newPosition(int~delta)~{
```

```
~~~~~~~~position~=~position~+~delta;
~~~~}
}
~
class~AParticle~extends~Particle~{
~~~~double~spin;
~
~~~~AParticle(int~p,~double~s)~{
~~~~~~~~super(p);
~~~~~~~~spin~=~s;
~~~~}
~
~~~~void~newPosition(int~delta)~{
~~~~if~(spin~<~delta)
~~~~~~~~~~~~position~=~position~+~delta;
~~~~}
}
~
class~BParticle~extends~Particle~{
~~~~int~charge;
~
~~~~BParticle(int~p,~int~c)~{
~~~~~~~~super(p);
~~~~~~~~charge~=~c;
~~~~}
}
~
class~CParticle~extends~BParticle~{
~~~~boolean~strange;
~
~~~~CParticle(int~p,~int~c,~boolean~s)~{
~~~~~~~~super(p,~c);
~~~~~~~~strange~=~s;
~~~~}
~
~~~~void~newPosition(int~delta)~{
~~~~~~~~if~(strange)
~~~~~~~~~~~~position~=~position~*~charge;
~~~~}
}
~
class~Inheritance01~{
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~Particle~~p~=~new~Particle(10);
~~~~~~~~AParticle~a~=~new~AParticle(20,~2.0);
~~~~~~~~BParticle~b~=~new~BParticle(30,~3);
~~~~~~~~CParticle~c~=~new~CParticle(40,~4,~true);
~
~~~~~~~~int~pPosition~=~p.position;
~~~~~~~~int~aPosition~=~a.position;
~~~~~~~~double~aSpin~~=~a.spin;
```

```
~~~~~~~~int~bPosition~=~b.position;
~~~~~~~~int~bCharge~=~b.charge;
~~~~~~~~int~cPosition~=~c.position;
~~~~~~~~int~cCharge~=~c.charge;
~~~~~~~~boolean~cStrange~=~c.strange;
~~~~}
}
```

In this simple program objects of all four classes are created and their fields are read.

- The objects are created.
- For each field of each object, a variable is declared in the main method and the value of the field is assigned to it. Check that each value originates from the correct field.

Exercise In `CParticle`, add the declaration `int charge = -1;`. Compile and run the program. Is the output different? Explain what happens.

6.1.2 Inheriting methods

Concept Subclasses inherit the methods of its superclasses and can add new methods of its own. You can override an inherited method by writing a new method with *the same signature* as the inherited method.

Program: Inheritance02.java

```
    //~Learning~Object~Inheritance02
//~~~~inheriting~and~overriding~methods
class~Particle~{
~~~~int~position;
~
~~~~Particle(int~p)~{
~~~~~~~~position~=~p;
~~~~}
~
~~~~void~newPosition(int~delta)~{
~~~~~~~~position~=~position~+~delta;
~~~~}
}
~
class~AParticle~extends~Particle~{
~~~~double~spin;
~
~~~~AParticle(int~p,~double~s)~{
~~~~~~~~super(p);
~~~~~~~~spin~=~s;
~~~~}
~
~~~~void~newPosition(int~delta)~{
~~~~if~(spin~<~delta)
~~~~~~~~~~~~position~=~position~+~delta;
~~~~}
}
class~BParticle~extends~Particle~{
~~~~int~charge;
~
```

```
~~~~BParticle(int~p,~int~c)~{
~~~~~~~~super(p);
~~~~~~~~charge~=~c;
~~~~}
}
class~CParticle~extends~BParticle~{
~~~~boolean~strange;
~
~~~~CParticle(int~p,~int~c,~boolean~s)~{
~~~~~~~~super(p,~c);
~~~~~~~~strange~=~s;
~~~~}
~
~~~~void~newPosition(int~delta)~{
~~~~~~~~if~(strange)
~~~~~~~~~~~~position~=~position~*~charge;
~~~~}
}
~
class~Inheritance02~{
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~Particle~~p~=~new~Particle(10);
~~~~~~~~AParticle~a~=~new~AParticle(10,~2.0);
~~~~~~~~BParticle~b~=~new~BParticle(10,~3);
~~~~~~~~CParticle~c~=~new~CParticle(10,~4,~true);
~
~~~~~~~~p.newPosition(10);
~~~~~~~~int~pPosition~=~p.position;
~~~~~~~~a.newPosition(10);
~~~~~~~~int~aPosition~=~a.position;
~~~~~~~~b.newPosition(10);
~~~~~~~~int~bPosition~=~b.position;
~~~~~~~~c.newPosition(10);
~~~~~~~~int~cPosition~=~c.position;
~~~~}
}
```

This program calls the method `newPosition`, which is overridden in `AParticle` and `CParticle` but not in `BParticle`.

- The objects are created.
- Method `newPosition` is invoked for each object and the modified value of `position` is assigned to a variable.
- Check that the call on `p` calls the method defined in class `Particle`.
- Check that the call on `a` calls the method defined in the class `AParticle`; this method overrides the method declared in class `Particle`.
- Check that the call on `b` calls the method defined in the superclass `Particle`; since the method was *not* overridden in `BParticle`, the method called is the one inherited from the superclass.
- Check that the call on `c` calls the method defined in the class `BParticle`; this method overrides the method declared in class `Particle`.

Exercise Remove the method `newPosition` from `CParticle`. Which method is invoked for `c.newPosition`?

Exercise Remove the method `newPosition` from `CParticle` and add a method with the same signature to `BParticle`. Which method is invoked for `c.newPosition`?

6.1.3 Dynamic dispatching

Concept A variable `v` of type `T` can contain a reference to an object of type `T` or of the type of any subclass of `T`. When invoking `v.m` for some method `m` that is overridden in a subclass, it is the type of the *object* currently referenced by `v` (not the type of the *variable*`v`) that determines which method is called. This is called *dynamic dispatching* because the call is dispatched at runtime.

Program: Inheritance03.java

```
    //~Learning~Object~Inheritance03
//~~~~dynamic~dispatching
class~Particle~{
~~~~int~position;
~
~~~~Particle(int~p)~{
~~~~~~~~position~=~p;
~~~~}
~
~~~~void~newPosition(int~delta)~{
~~~~~~~~position~=~position~+~delta;
~~~~}
}
~
class~AParticle~extends~Particle~{
~~~~double~spin;
~
~~~~AParticle(int~p,~double~s)~{
~~~~~~~~super(p);
~~~~~~~~spin~=~s;
~~~~}
~
~~~~void~newPosition(int~delta)~{
~~~~if~(spin~<~delta)
~~~~~~~~~~~~position~=~position~+~delta;
~~~~}
}
~
class~BParticle~extends~Particle~{
~~~~int~charge;
~
~~~~BParticle(int~p,~int~c)~{
~~~~~~~~super(p);
~~~~~~~~charge~=~c;
~~~~}
}
~
class~CParticle~extends~BParticle~{
~~~~boolean~strange;
~
~~~~CParticle(int~p,~int~c,~boolean~s)~{
```

```
~~~~~~~~super(p,~c);
~~~~~~~~strange~=~s;
~~~~}
~
~~~~void~newPosition(int~delta)~{
~~~~~~~~if~(strange)
~~~~~~~~~~~~position~=~position~*~charge;
~~~~}
}
~
class~Inheritance03~{
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~Particle~~p~=~new~Particle(10);
~~~~~~~~AParticle~a~=~new~AParticle(20,~2.0);
~~~~~~~~BParticle~b~=~new~BParticle(30,~3);
~~~~~~~~CParticle~c~=~new~CParticle(40,~4,~true);
~
~~~~~~~~p.newPosition(10);
~~~~~~~~int~pPosition~=~p.position;
~~~~~~~~p~=~a;
~~~~~~~~p.newPosition(10);
~~~~~~~~int~aPosition~=~p.position;
~~~~~~~~p~=~b;
~~~~~~~~p.newPosition(10);
~~~~~~~~int~bPosition~=~p.position;
~~~~~~~~p~=~c;
~~~~~~~~p.newPosition(10);
~~~~~~~~int~cPosition~=~p.position;
~~~~}
}
```

- The objects are created.
- The references to the objects are assigned one-by-one to the variable `p`, and then the method `newPosition` is invoked using `p`. The modified value of `position` is assigned to a variable.
- Check that the call on `p` invokes the method defined in class `Particle`.
- After assigning `a` to `p`, check that the call invokes the method defined in the class `AParticle`; this method overrides the method declared in class `Particle`. Although the type of `p` is class `Particle`, it holds a reference to an object whose type is class `AParticle` so the method of that class is called.
- Note that as a result of the assignment, the object of type `Particle` has become garbage.
- After assigning `b` to `p`, check that the call invokes the method defined in the superclass `Particle`; since the method was *not* overridden in `BParticle`, the method called is the one inherited from the superclass.
- After assigning `c` to `p`, check that the call invokes the method defined in the class `CParticle`; this method overrides the method declared in class `Particle`. Although the type of `p` is class `Particle`, it holds a reference to an object whose type is class `CParticle` so the method of that class is called.

Exercise Add an assignment of `c` to `b` and call `b.newPosition(10)`. What is the value now of `b.position`?

6.1.4 Downcasting

Concept A variable of the type of a class can reference an object of the type of a *subclass*, but this variable cannot be used to access fields declared in the subclass. Nevertheless, the object “remembers” its type, even if it is assigned to a variable of the type of its superclass, and the type can be "recovered" by casting to a variable of the type of the subclass. This is called *downcasting* because the cast is "down" the derivation hierarchy.

Program: Inheritance04.java

```
    //~Learning~Object~Inheritance04
//~~~~downcasting
class~Particle~{
~~~~int~position;
~
~~~~Particle(int~p)~{
~~~~~~~~position~=~p;
~~~~}
~
~~~~void~newPosition(int~delta)~{
~~~~~~~~position~=~position~+~delta;
~~~~}
}
~
class~AParticle~extends~Particle~{
~~~~double~spin;
~
~~~~AParticle(int~p,~double~s)~{
~~~~~~~~super(p);
~~~~~~~~spin~=~s;
~~~~}
~
~~~~void~newPosition(int~delta)~{
~~~~if~(spin~<~delta)
~~~~~~~~~~~~position~=~position~+~delta;
~~~~}
}
~
class~BParticle~extends~Particle~{
~~~~int~charge;
~
~~~~BParticle(int~p,~int~c)~{
~~~~~~~~super(p);
~~~~~~~~charge~=~c;
~~~~}
}
~
class~CParticle~extends~BParticle~{
~~~~boolean~strange;
~
~~~~CParticle(int~p,~int~c,~boolean~s)~{
~~~~~~~~super(p,~c);
~~~~~~~~strange~=~s;
~~~~}
~
~~~~void~newPosition(int~delta)~{
~~~~~~~~if~(strange)
```

```
~~~~~~~~~~~~position~=~position~*~charge;
~~~~}
}
~
class~Inheritance04~{
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~Particle~~p~=~new~Particle(10);
~~~~~~~~AParticle~a~=~new~AParticle(20,~2.0);
~
~~~~~~~~int~pPosition~=~p.position;
~~~~~~~~p~=~a;
~~~~~~~~int~paPosition~=~p.position;
~~~~~~~~a~=~(AParticle)~p;
~~~~~~~~int~aPosition~=~a.position;
~~~~~~~~double~sSpin~=~a.spin;
~~~~}
}
```

In this program, we take an object of the type of the subclass `AParticle` and assign its reference to the variable `p` of the type of the superclass `Particle`. The object's actual type is recovered by downcasting from `p` to `a`.

- The objects are created.
- The value of `p.position` is stored in a variable.
- The reference in the variable `a` is assigned to `p`. Note that the arrows from the representation of both variables point to the same object of type `AParticle`, and that the other object is garbage.
- When the value of `p.position` is accessed, it refers to the value that is in `a.position`.
- The reference in `p` can be cast to the type `AParticle` and assigned to `a`. Although `p` is declared to hold references to objects of type `Particle`, the object was really of the subclass `AParticle`.
- Both the fields `position` and `spin` can be accessed through `a`.

Exercise What happens if you try to access `p.spin` after `a` has been assigned to `p`?

Exercise Add the statement `BParticle b = (BParticle) p` after the assignment of `a` to `p`. Does the program compile successfully? Does it run successfully? Explain the results.

6.1.5 Heterogeneous data structures

Concept A heterogeneous data structure is one that can hold elements of different types. A data structure whose elements are of the type of a class can hold references to objects of any subclass of that class.

Program: Inheritance05.java

```
    //~Learning~Object~Inheritance05
//~~~~heterogeneous~data~structures
class~Particle~{
~~~~int~position;
~
~~~~Particle(int~p)~{
~~~~~~~~position~=~p;
~~~~}
~
~~~~void~newPosition(int~delta)~{
~~~~~~~~position~=~position~+~delta;
~~~~}
```

```
}
~
class~AParticle~extends~Particle~{
~~~~double~spin;
~
~~~~AParticle(int~p,~double~s)~{
~~~~~~~~super(p);
~~~~~~~~spin~=~s;
~~~~}
~
~~~~void~newPosition(int~delta)~{
~~~~if~(spin~<~delta)
~~~~~~~~~~~~position~=~position~+~delta;
~~~~}
}
~
class~BParticle~extends~Particle~{
~~~~int~charge;
~
~~~~BParticle(int~p,~int~c)~{
~~~~~~~~super(p);
~~~~~~~~charge~=~c;
~~~~}
}
~
class~CParticle~extends~BParticle~{
~~~~boolean~strange;
~
~~~~CParticle(int~p,~int~c,~boolean~s)~{
~~~~~~~~super(p,~c);
~~~~~~~~strange~=~s;
~~~~}
~
~~~~void~newPosition(int~delta)~{
~~~~~~~~if~(strange)
~~~~~~~~~~~~position~=~position~*~charge;
~~~~}
}
~
class~Inheritance05~{
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~Particle[]~p~=~new~Particle[4];
~~~~~~~~p[0]~=~new~Particle(10);
~~~~~~~~p[1]~=~new~AParticle(20,~2.0);
~~~~~~~~p[2]~=~new~BParticle(30,~3);
~~~~~~~~p[3]~=~new~CParticle(40,~4,~true);
~~~~~~~~int~i~=~0;
~~~~~~~~p[i++].newPosition(10);
~~~~~~~~p[i++].newPosition(10);
~~~~~~~~p[i++].newPosition(10);
~~~~~~~~p[i].newPosition(10);
```

```
~~~~}
}
```

An array whose elements are of class `Particle` can store references to objects of any of its subclasses.

- The objects are created and references to them assigned to elements of the elements of the array `p`.
- Method `newPosition` is invoked for the object referenced by each element of the array `p`. Check that the fields accessed are those of the object referenced by the array element and that the calls are dynamically dispatched to the method appropriate for the type of the object.

Exercise Every object in Java is a subclass of the class `Object`. Modify the program so that the variable `p` is of type array of `Object`.

6.1.6 Abstract classes

Concept Very often the "root" of a set of derived types has no meaning itself, in the sense that objects of that type would never be declared. For example, in a realistic simulation program, the would be no real particles that are just "particles," only particles with names like α-particles and β-particles. An *abstract* class can be declared which serves only as a root from which to derive a hierarchy of subclasses. It is not legal to declare *objects* of an abstract class, although *variables* of its type may be declared and used to reference objects of any type within the hierarchy. A method may also be declared abstract; this indicates that it *must* be overridden in subclasses.

Program: Inheritance06.java

```
    //~Learning~Object~Inheritance06
//~~~~abstract~classes
abstract~class~Particle~{
~~~~int~position;
~
~~~~Particle(int~p)~{
~~~~~~~~position~=~p;
~~~~}
~~~~abstract~void~newPosition(int~delta);
}
~
class~AParticle~extends~Particle~{
~~~~double~spin;
~
~~~~AParticle(int~p,~double~s)~{
~~~~~~~~super(p);
~~~~~~~~spin~=~s;
~~~~}
~
~~~~void~newPosition(int~delta)~{
~~~~if~(spin~<~delta)
~~~~~~~~~~~~position~=~position~+~delta;
~~~~}
}
~
class~BParticle~extends~Particle~{
~~~~int~charge;
~
~~~~BParticle(int~p,~int~c)~{
```

```
~~~~~~~~super(p);
~~~~~~~~charge~=~c;
~~~~}
}
~
class~CParticle~extends~BParticle~{
~~~~boolean~strange;
~
~~~~CParticle(int~p,~int~c,~boolean~s)~{
~~~~~~~~super(p,~c);
~~~~~~~~strange~=~s;
~~~~}
~
~~~~void~newPosition(int~delta)~{
~~~~~~~~if~(strange)
~~~~~~~~~~~~position~=~position~*~charge;
~~~~}
}
~
class~Inheritance06~{
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~Particle[]~p~=~new~Particle[3];
~~~~~~~~p[0]~=~new~AParticle(20,~2.0);
~~~~~~~~p[1]~=~new~BParticle(30,~3);
~~~~~~~~p[2]~=~new~CParticle(40,~4,~true);
~~~~~~~~int~i~=~0;
~~~~~~~~p[i++].newPosition(10);
~~~~~~~~p[i++].newPosition(10);
~~~~~~~~p[i].newPosition(10);
~~~~}
}
```

The follow program declares `Particle` to be `abstract` and no objects of that class can be declared. The method `newPosition` is also declared abstrct because it doesn't make sense to have a particle that you can't move.

Exercise The program does not compile successfully. Why? (Note that in Jeliot, the problem is only found at when animating the program.) Modify the program so that it compiles and executes.

- The objects are created and references to them assigned to elements of the array.
- Method `newPosition` is invoked for the object referenced by each element of the array `p`. Check that the fields accessed are those of the object referenced by the array element and that the calls are dynamically dispatched to the method appropriate for the type of the object.

Exercise It is possible to declare a nonabsract method in an abstract class. Give an example for this program, and explain why it is a reasonable thing to do.

6.1.7 Equals

Concept There are two concepts of equality in Java: the *operator*`==` compares primitives types and references, while the *method*`equals` compares objects. The default implementation of `equals` is like `==`, but it can be overridden in any class.

Program: Inheritance07A.java

```
    //~Learning~Object~Inheritance07A
//~~~~equality~(==~vs.~equals)
class~Particle~{
~~~~int~position;
~
~~~~Particle(int~p)~{
~~~~~~~~position~=~p;
~~~~}
~
~~~~void~newPosition(int~delta)~{
~~~~~~~~position~=~position~+~delta;
~~~~}
}
~
class~AParticle~extends~Particle~{
~~~~double~spin;
~
~~~~AParticle(int~p,~double~s)~{
~~~~~~~~super(p);
~~~~~~~~spin~=~s;
~~~~}
~
~~~~void~newPosition(int~delta)~{
~~~~if~(spin~<~delta)
~~~~~~~~~~~~position~=~position~+~delta;
~~~~}
}
~
class~Inheritance07A~{
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~AParticle~a1~=~new~AParticle(20,~2.0);
~~~~~~~~AParticle~a2~=~a1;
~~~~~~~~AParticle~a3~=~new~AParticle(20,~2.0);
~~~~~~~~boolean~eqop12~=~a1~==~a2;
~~~~~~~~boolean~eqop13~=~a1~==~a3;
~~~~~~~~boolean~eqmethod~=~a1.equals(a3);
~~~~}
}
```

- Object `a1` of type `AParticle` is created.
- `a1` is assigned to `a2` using `==`.
- Object `a3` of type `AParticle` is created with the same values for its fields as the object referenced by `a1`.
- Evaluating `a1==a2` returns *true* because they both reference the same object.
- Evaluating `a1==a3` returns *false* because they reference different objects.
- Strangely enough, evaluating `a1.equals(a3)` returns *false*. Although their fields are equal, the default implementation of `equals` is the same as `==`!

Exercise Add the follow method to `AParticle` and run the program again. What happens now?

```
    public~boolean~equals(AParticle~a)~{
~~return~this.position~==~a.position~&&~this.spin~==~a.spin;
}
```

Program: Inheritance07B.java

```
    //~Learning~Object~Inheritance07B
//~~~~equality~(overloading~equals)
class~Particle~{
~~~~int~position;
~
~~~~Particle(int~p)~{
~~~~~~~~position~=~p;
~~~~}
~
~~~~void~newPosition(int~delta)~{
~~~~~~~~position~=~position~+~delta;
~~~~}
}
~
class~BParticle~extends~Particle~{
~~~~int~charge;
~
~~~~BParticle(int~p,~int~c)~{
~~~~~~~~super(p);
~~~~~~~~charge~=~c;
~~~~}
~
~~~~public~boolean~equals(BParticle~b)~{
~~~~~~~~return~~this.position~==~b.position~&&
~~~~~~~~~~~~~~~~this.charge~==~b.charge;
~~~~}
}
~
class~CParticle~extends~BParticle~{
~~~~boolean~strange;
~
~~~~CParticle(int~p,~int~c,~boolean~s)~{
~~~~~~~~super(p,~c);
~~~~~~~~strange~=~s;
~~~~}
~
~~~~void~newPosition(int~delta)~{
~~~~~~~~if~(strange)
~~~~~~~~~~~~position~=~position~*~charge;
~~~~}
~
~~~~public~boolean~equals(CParticle~c)~{
~~~~~~~~return~~~~this.position~==~c.position~&&
~~~~~~~~~~~~~~~~this.charge~==~c.charge~&&
~~~~~~~~~~~~~~~~~~this.strange~==~c.strange;
~~~~}
}
~
class~Inheritance07B~{
~~~~public~static~void~main(/*String[]~args*/)~{
```

```
~~~~~~~~BParticle~b1~=~new~BParticle(20,~2);
~~~~~~~~BParticle~b2~=~new~BParticle(20,~2);
~~~~~~~~CParticle~c1~=~new~CParticle(20,~2,~false);
~~~~~~~~CParticle~c2~=~new~CParticle(20,~2,~true);
~~~~~~~~boolean~eqb1b2~=~b1.equals(b2);
~~~~~~~~boolean~eqc1c2~=~c1.equals(c2);
~~~~~~~~boolean~eqb1c1~=~b1.equals(c1);
~~~~~~~~boolean~eqc1b1~=~c1.equals(b1);
~~~~}
}
```

Let us try to override the method `equals` in classes `BParticle` and `CParticle`; the method returns true if the all fields of the two objects are equal.

- Four objects are created: two equal objects `b1` and `b2` of type `BParticle` and two unequal objects `c1` and `c2` of type `CParticle`.
- As expected, `b1.equals(b2)` returns *true* and `c1.equals(c2)` returns *false*.
- `b1.equals(c1)` returns *true*: since `CParticle` is a subclass of `BParticle`, the variable `c1` is acceptable as a parameter to the method `equals` declared in `BParticle`. `c1`*is* equal to `b1`, because we are only comparing the first two fields inherited from `BParticle` and these are equal.

Exercise Explain what happens if you try to evaluate `c1.equals(b1)`.

Program: Inheritance07C.java

```
    //~Learning~Object~Inheritance07C
//~~~~equality~(robust~overriding)
class~Particle~{
~~~~int~position;
~
~~~~Particle(int~p)~{
~~~~~~~~position~=~p;
~~~~}
~
~~~~void~newPosition(int~delta)~{
~~~~~~~~position~=~position~+~delta;
~~~~}
}
~
class~BParticle~extends~Particle~{
~~~~int~charge;
~
~~~~BParticle(int~p,~int~c)~{
~~~~~~~~super(p);
~~~~~~~~charge~=~c;
~~~~}
}
~
class~CParticle~extends~BParticle~{
~~~~boolean~strange;
~
~~~~CParticle(int~p,~int~c,~boolean~s)~{
~~~~~~~~super(p,~c);
~~~~~~~~strange~=~s;
```

```
~~~~}
~
~~~~void~newPosition(int~delta)~{
~~~~~~~~if~(strange)
~~~~~~~~~~~~position~=~position~*~charge;
~~~~}
~
~~~~public~boolean~equals(Object~obj)~{
~~~~~~~~if~(obj~==~null)~return~false;
~~~~~~~~if~(!(obj~instanceof~CParticle))~return~false;
~~~~~~~~CParticle~c~=~(CParticle)~obj;
~~~~~~~~return~this.position~==~c.position~&&~this.charge~==~c.charge~&&
~~~~~~~~~~~~~~~this.strange~==~c.strange;
~~~~}
}
~
class~Inheritance07C~{
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~BParticle~b1~=~new~BParticle(20,~2);
~~~~~~~~CParticle~c1~=~new~CParticle(20,~2,~false);
~~~~~~~~CParticle~c2~=~new~CParticle(20,~2,~true);
~~~~~~~~CParticle~c3~=~new~CParticle(20,~2,~false);
~~~~~~~~boolean~eqc1null~=~c1.equals(null);
~~~~~~~~boolean~eqc1b1~=~c1.equals(b1);
~~~~~~~~boolean~eqc1c2~=~c1.equals(c2);
~~~~~~~~boolean~eqc1c3~=~c1.equals(c3);
~~~~}
}
```

It would be unusual for two objects to be considered equal if they are of different types, even if one type is a subclass of another. In fact, `public boolean equals(CParticle c)` does not override the method `equals` in `BParticle`, because an overriding method must have the *same signature* as the overridden method.

The method `equals` is declared in the root class `Object` as: `public boolean equals(Object obj)` and this is the method that must be overridden. This program shows the correct technique:

- Since the parameter can now be any object, a check is first made that the parameter is not `null`.
- Similarly, a check is made that the parameter is of the same type as this object.
- Now that we know that the parameter is actually of this type, it can be cast from `Object` to the type.
- Only then is class-specific code performed—usually a field-by-field comparison.

Trace the execution of the program:

- Four objects are created: one object `b1` of type `BParticle` and three objects `c1`, `c2` and `c3` of type `CParticle`.
- Clearly, comparing `c1` to `null` or `b1` returns *false*.
- Field-by-field comparisons are used if the parameter is of type `CParticle`: `c1.equals(c2)` returns *false* and `c1.equals(c3)` returns *true*.

Exercise Move the declaration of `equals` to class `BParticle`, changing the code as needed. What now is the value of `c1.equals(b1)`? Explain.

6.1.8 Clone

Concept Assigning a variable containing a reference to another of the same type merely copies the reference so that two fields refer to the same object. The method `clone` is used to copy the content of an object into a new one. `clone` is defined in class `Object` and can be overridden in any class definition.

Program: Inheritance08.java

```
    //~Learning~Object~Inheritance08
//~~~~clone
class~Particle~implements~Cloneable~{
~~~~int~position;
~
~~~~Particle(int~p)~{
~~~~~~~~position~=~p;
~~~~}
~
~~~~void~newPosition(int~delta)~{
~~~~~~~~position~=~position~+~delta;
~~~~}
~
~~~~protected~Object~clone()~{
~~~~~~~~try~{
~~~~~~~~~~~~Particle~p~=~(Particle)~super.clone();
~~~~~~~~~~~~p.newPosition(10);
~~~~~~~~~~~~return~p;
~~~~~~~~}
~~~~~~~~catch~(CloneNotSupportedException~e)~{
~~~~~~~~~~~~e.printStackTrace();
~~~~~~~~~~~~throw~new~Error();
~~~~~~~~}
~~~~}
}
~
class~Inheritance08~{
~~~~public~static~void~main(String[]~args)~{
~~~~~~~~Particle~p1~=~new~Particle(20);
~~~~~~~~Particle~p2~=~p1;
~~~~~~~~p1.newPosition(10);
~~~~~~~~System.out.println(p1.position);
~~~~~~~~System.out.println(p2.position);
~
~~~~~~~~Particle~p3~=~(Particle)~p1.clone();
~~~~~~~~System.out.println(p1.position);
~~~~~~~~System.out.println(p3.position);
~
~~~~~~~~p3.newPosition(10);
~~~~~~~~System.out.println(p1.position);
~~~~~~~~System.out.println(p3.position);
~~~~}
}
```

`clone` is overridden in class `Particle`. The class must implement the interface `Cloneable`, the method of the superclass should be called, and we have to take into account that the method might raise an exception.

The method returns the object returned by superclass method after calling `newPosition`.

- An object of class `Particle` is allocated and its reference assigned to the field `p1`.
- An assignment statement copies this reference to the field `p2`. Check that they have the same value.
- The method `newPosition` is called on `p1`, but the value of `p2.position` is also changed, showing that the two fields point to the same object.
- An object of class `Particle` is obtained by calling `p1.clone()` and its reference assigned to the field `p3`. Since `clone` returns a value of type `Object`, it must be cast to type `Particle` before the assignment. Check that the objects referenced by `p1` and `p3` have different values.
- Calling `p3.newPosition` changes only the field in the object referenced by `p3` and not the separate object referenced by `p1`.

Exercise The method `clone` can perform arbitrary computation. Modify the program so that new objects are initialized with the absolute value of the field of the object that is being cloned.

6.1.9 Overloading vs. overriding

Concept*Overloading* is the use of the same method name with a *different* parameter signature. *Overriding* is the use in a subclass of the same method name with the *same* parameter signature as a method of the superclass.

Program: Inheritance09.java

```
    //~Learning~Object~Inheritance09
//~~~~overloading~vs.~overriding
class~Particle~{
~~~~int~position;
~
~~~~Particle(int~p)~{
~~~~~~~~position~=~p;
~~~~}
~
~~~~void~newPosition(int~delta)~{
~~~~~~~~position~=~position~+~delta;
~~~~}
}
~
class~AParticle~extends~Particle~{
~~~~double~spin;
~
~~~~AParticle(int~p,~double~s)~{
~~~~~~~~super(p);
~~~~~~~~spin~=~s;
~~~~}
~
~~~~void~newPosition(int~delta)~{
~~~~if~(spin~<~delta)
~~~~~~~~~~~~position~=~position~+~delta;
~~~~}
~
~~~~void~newPosition(double~delta)~{
~~~~if~(position~<~delta)
~~~~~~~~~~~~spin~=~spin~+~delta;
~~~~}
```

```
}
~
class~Inheritance09~{
~~~~public~static~void~main(/*String[]~args*/)~{
~~~~~~~~Particle~~p~=~new~Particle(10);
~~~~~~~~AParticle~a1~=~new~AParticle(20,~-1.0);
~~~~~~~~AParticle~a2~=~new~AParticle(20,~-1.0);
~
~~~~~~~~p.newPosition(10);
~~~~~~~~int~pPosition~=~p.position;
~~~~~~~~a1.newPosition(10);
~~~~~~~~int~a1Position~=~a1.position;
~~~~~~~~a2.newPosition(10.0);
~~~~~~~~int~a2Position~=~a2.position;
~~~~}
}
```

The method `newPosition(int delta)` is declared in `Particle` and *overridden* in `AParticle`. It is also *overloaded* by a method with the same name takes a parameter of type `double`.

- After allocating three objects `p`, `a1` and `a2`, `newPosition` is called on each one.
- `p.newPosition` calls the method declared in class `Particle`.
- `a1.newPosition` calls the method declared in class `AParticle` that overrides the method in `Particle`.
- `a2.newPosition` calls the overloaded method because the actual parameter is of type `double`.

Exercise At the end of the program add an assignment `p = a1`. Add the method invocations `p.newPosition(10)` and `p.newPosition(10.0)` in the main method. Explain what happens.

Index of Keywords and Terms

Keywords are listed by the section with that keyword (page numbers are in parentheses). Keywords do not necessarily appear in the text of the page. They are merely associated with that section. *Ex.* apples, § 1.1 (1) **Terms** are referenced by the page they appear on. *Ex.* apples, 1

Attributions

Collection: *Learning Objects for Java (with Jeliot)*
Edited by: Mordechai (Moti) Ben-Ari
URL: http://cnx.org/content/col10915/1.2/
License: http://creativecommons.org/licenses/by/3.0/

Module: "Learning Objects for Java (Overview)"
By: Mordechai (Moti) Ben-Ari
URL: http://cnx.org/content/m31242/1.3/
Pages: 1-2
Copyright: Mordechai (Moti) Ben-Ari
License: http://creativecommons.org/licenses/by/3.0/

Module: "Learning Objects for Control Structures in Java"
By: Mordechai (Moti) Ben-Ari
URL: http://cnx.org/content/m31246/1.1/
Pages: 3-11
Copyright: Mordechai (Moti) Ben-Ari
License: http://creativecommons.org/licenses/by/3.0/

Module: "Learning Objects for Methods in Java"
By: Mordechai (Moti) Ben-Ari
URL: http://cnx.org/content/m31247/1.1/
Pages: 13-24
Copyright: Mordechai (Moti) Ben-Ari
License: http://creativecommons.org/licenses/by/3.0/

Module: "Learning Objects for Arrays in Java"
By: Mordechai (Moti) Ben-Ari
URL: http://cnx.org/content/m31245/1.1/
Pages: 25-33
Copyright: Mordechai (Moti) Ben-Ari
License: http://creativecommons.org/licenses/by/3.0/

Module: "Learning Objects for Constructors in Java"
By: Mordechai (Moti) Ben-Ari
URL: http://cnx.org/content/m31248/1.1/
Pages: 35-48
Copyright: Mordechai (Moti) Ben-Ari
License: http://creativecommons.org/licenses/by/3.0/

Module: "Learning Objects for Inheritance in Java"
By: Mordechai (Moti) Ben-Ari
URL: http://cnx.org/content/m31249/1.1/
Pages: 49-67
Copyright: Mordechai (Moti) Ben-Ari
License: http://creativecommons.org/licenses/by/3.0/

Learning Objects for Java (with Jeliot)
This is a set of learning objects (LO) for studying introductory programming with Java. Each LO can be studied independently. They are divided into five modules on the topics: control structures, methods, arrays, constructors and inheritance. The LOs are designed for use with the Jeliot program animation system.

About Connexions
Since 1999, Connexions has been pioneering a global system where anyone can create course materials and make them fully accessible and easily reusable free of charge. We are a Web-based authoring, teaching and learning environment open to anyone interested in education, including students, teachers, professors and lifelong learners. We connect ideas and facilitate educational communities.

Connexions's modular, interactive courses are in use worldwide by universities, community colleges, K-12 schools, distance learners, and lifelong learners. Connexions materials are in many languages, including English, Spanish, Chinese, Japanese, Italian, Vietnamese, French, Portuguese, and Thai. Connexions is part of an exciting new information distribution system that allows for **Print on Demand Books**. Connexions has partnered with innovative on-demand publisher QOOP to accelerate the delivery of printed course materials and textbooks into classrooms worldwide at lower prices than traditional academic publishers.