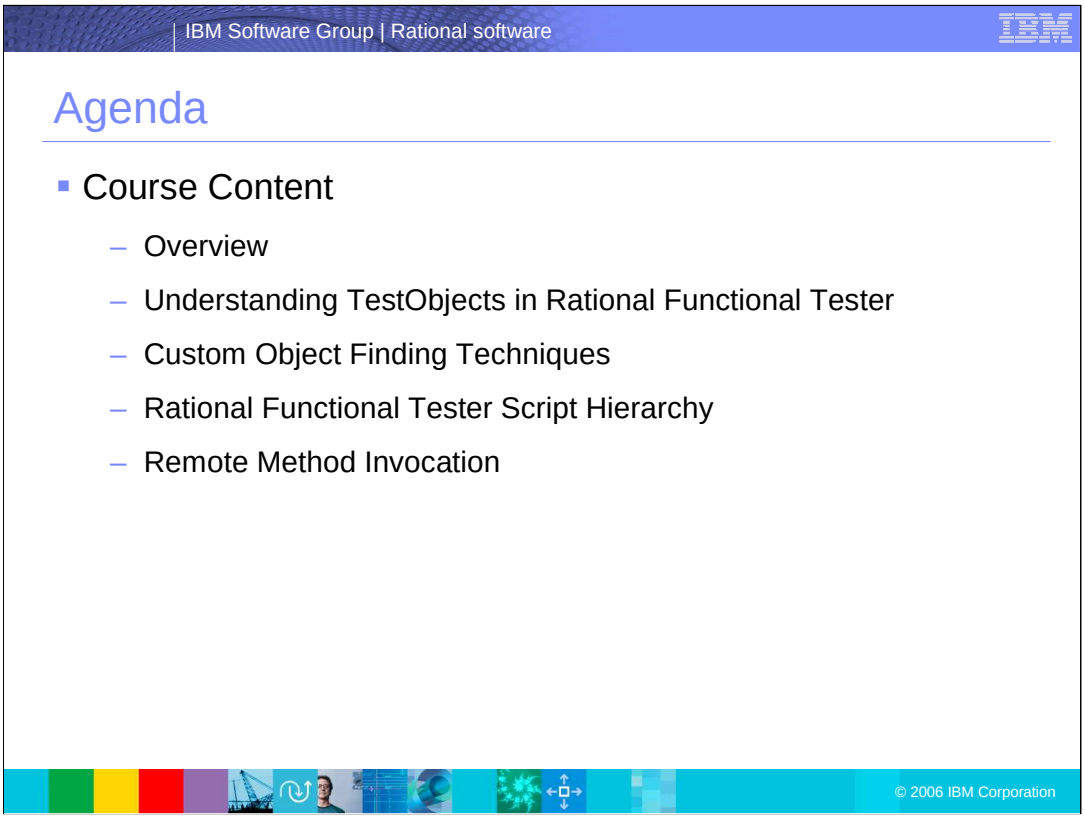
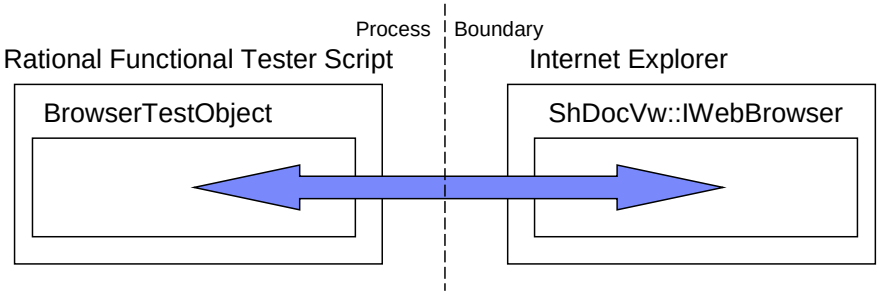
This module will cover some advanced scripting topics in Rational Functional Tester.



By the end of this module, you should understand TestObjects in Rational Functional Tester, custom object finding techniques, script hierarchy, and remote method invocation.

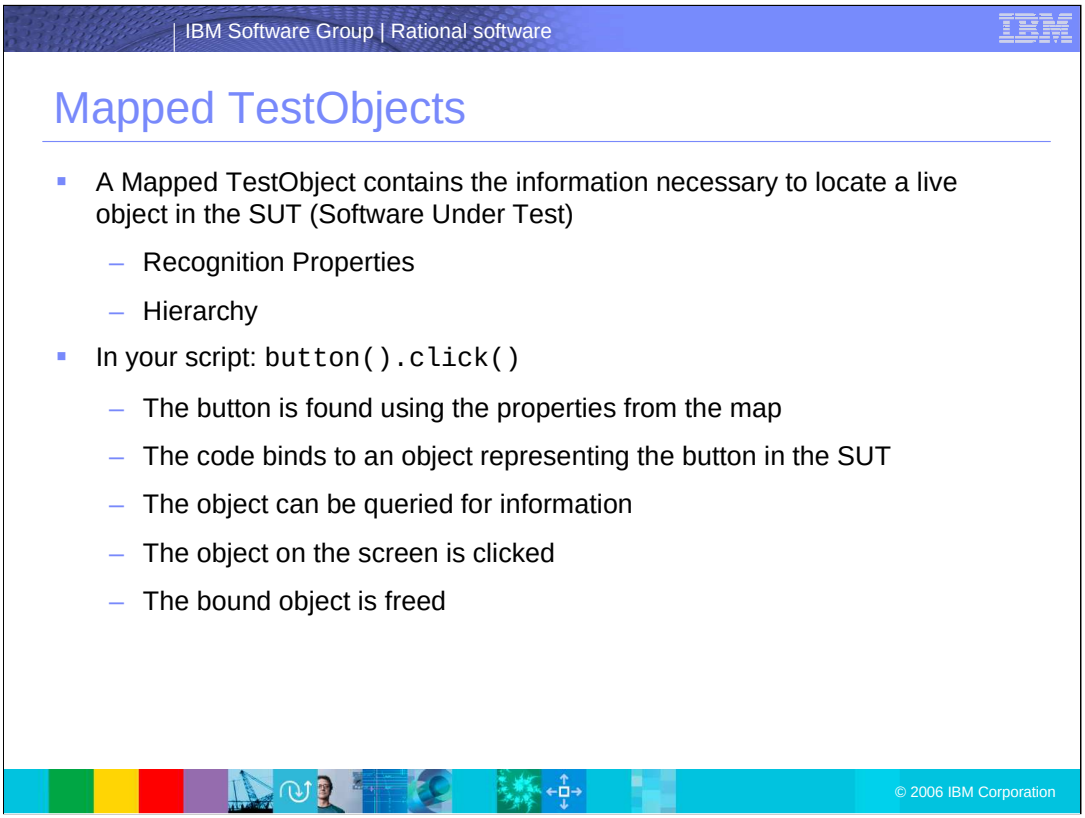
What is a TestObject?

- A TestObject represents a connection point between a Rational Functional Tester script and the Software Under Test (SUT).
  - Most methods called on a TestObject are invoked in the SUT
- Two categories of TestObjects:
  - Mapped
  - Bound (“found”, “pinned”, “registered”)



A TestObject represents a connection point between a Rational Functional Tester script and the Software Under Test, or SUT. Most methods called on a TestObject are invoked in the Software Under Test.

There are two categories of TestObjects: Mapped and Bound (also known as “found”, “pinned”, or “registered”)

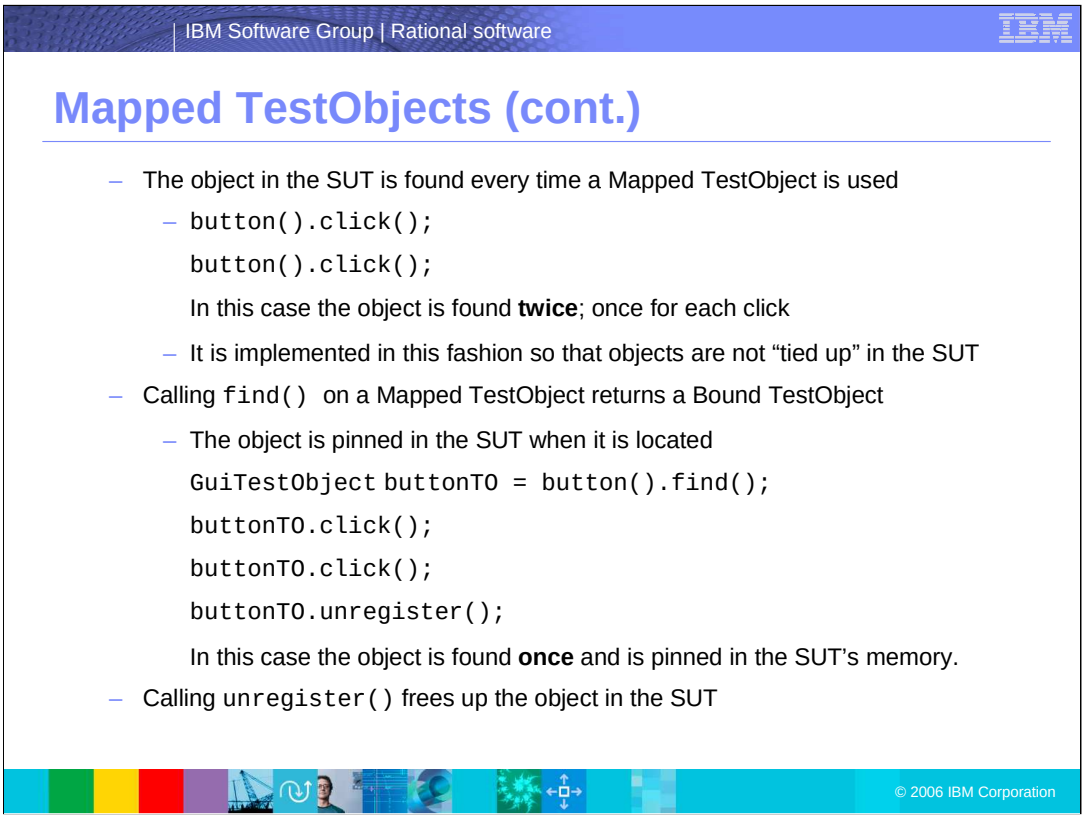


A Mapped TestObject contains the information necessary to locate a live object in the Software Under Test.

Recognition Properties are different properties of the TestObject being recognized by Rational Functional Tester.

The TestObject Hierarchy lists all test objects in the SUT and provides information on owner/owned relationships, states, test domains, and others.

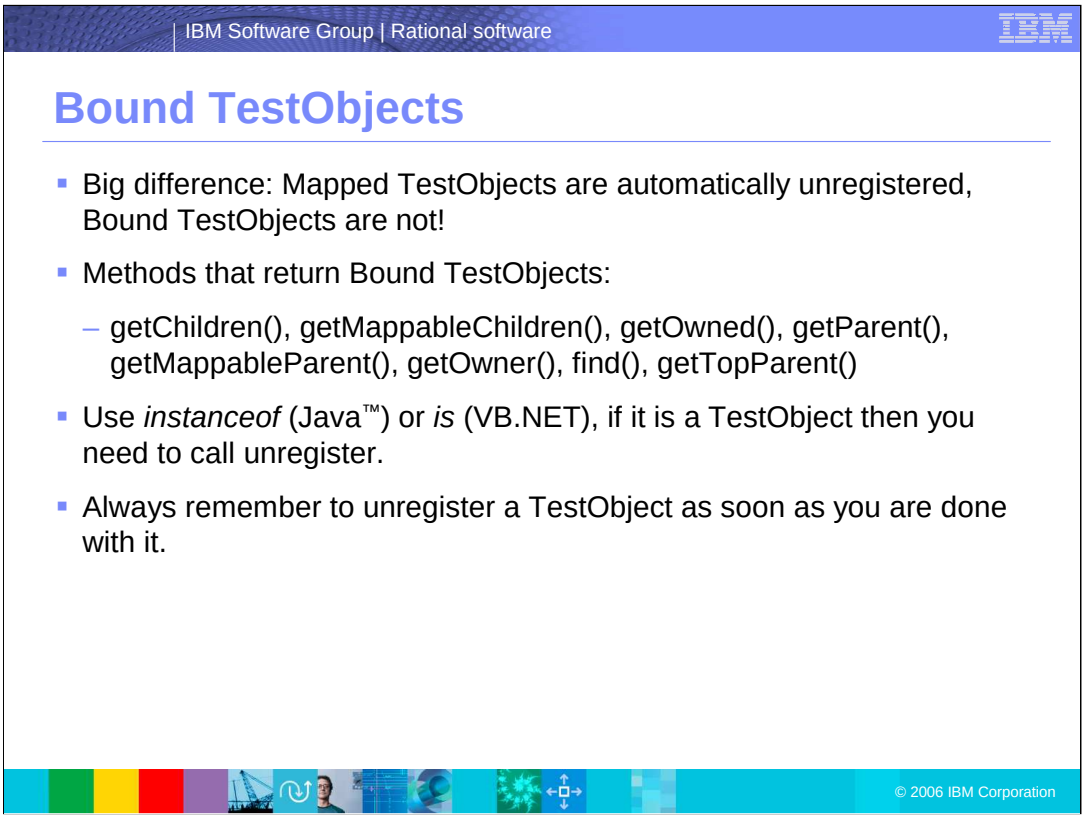
In your script: `button().click(),` the button is found using the properties from the map, and then bound to an object representing the button in the software under test. Next, object is queried for pertinent information, like screen coordinates, and so on. The object on the screen is clicked, and finally, the bound object is freed.



The object in the Software Under Test is found every time a Mapped TestObject is used. In this example, the object is found **twice**; once for each click. Objects are implemented in this fashion so that they are not “tied up” in the software under test.

Second, calling `find()` on a Mapped TestObject returns a Bound TestObject. The object is pinned in the Software Under Test when it is located. In this example, the object is found **once** and is pinned in the memory of the software under test.

Finally, calling `unregister()` frees up the object in the software under test.



When working with Bound TestObjects, there is a big difference between Mapped TestObjects and Bound TestObjects. Mapped TestObjects are automatically unregistered, while Bound TestObjects are not.

The methods shown here are used to return Bound TestObjects. As a hint, use the “instanceof” or “is” methods if it is a TestObject that you will eventually need to call unregister.

Always remember to unregister a TestObject as soon as you are done with it; remember, you are dealing with live objects.

Bound TestObjects (cont.)

- Some scripting hints:
  - Be careful about program flow!
  - Be careful about exceptions. It is best to put your call to `unregister` in a `finally` block.

```
TestObject TO = mappedObject.find();
try
{
        // what if the property does not exist?
        TO.getProperty("prop");
}
finally
{
        TO.unregister();
}
```


The example shows you how to unregister the bound object. It is helpful to put your call to unregister in a finally block as shown here.

Bound TestObjects (cont.)

- Some scripting hints:
  - A **great** tip for arrays is to null out the object you are interested in and call `unregister` on the whole array.

```
TestObject[] allResults = null;
TestObject myResult = null;
try
{
        allResults = MTO.getChildren();
        theGoodResult = allResults[0];
        allResults[0] = null;
 }
 finally
 {
        unregister(allResults);
}
```


Another scripting hint, when working with arrays, is to null out the object you are interested in and to call unregister on the whole array. In the example shown here, the unregister call unregisters the whole array through unregister(allResults);

The old way shown here invokes getChildren() and getProperties() multiple times, whereas, the new way invokes find() methods with the result of atDescendant method as a parameter instead.

There are some advantages of the `find()` API as opposed to a recursive search. It is:

1) *Much* faster - Every call to `getChildren()` and `getProperty()` goes cross process. `find()` goes cross process once, does the entire search, and returns,

2) *Much* safer - Very easy to leak registered objects in a custom search implementation, and

3) *Much* easier - `find()` syntax may be a little tricky at first but it is much easier than writing all the support code for custom solutions

Custom object finding techniques

- Two places to start the search:
  - The RootTestObject
    - The RootTestObject is the “world view” of the system. All domains will be searched.
    - `RootTestObject root = getRootTestObject();`
  - Any GUI TestObject, mapped or bound
    - Narrows down the focus of the search.
    - Less likely to get unexpected results.
    - The more you narrow the search, the faster it will run at playback time.

You can start the search from the root or from a specific GUI TestObject. The second option is preferred because it narrows down the focus of the search, is less likely to get unexpected results, and will run faster at playback time.

Custom object finding techniques

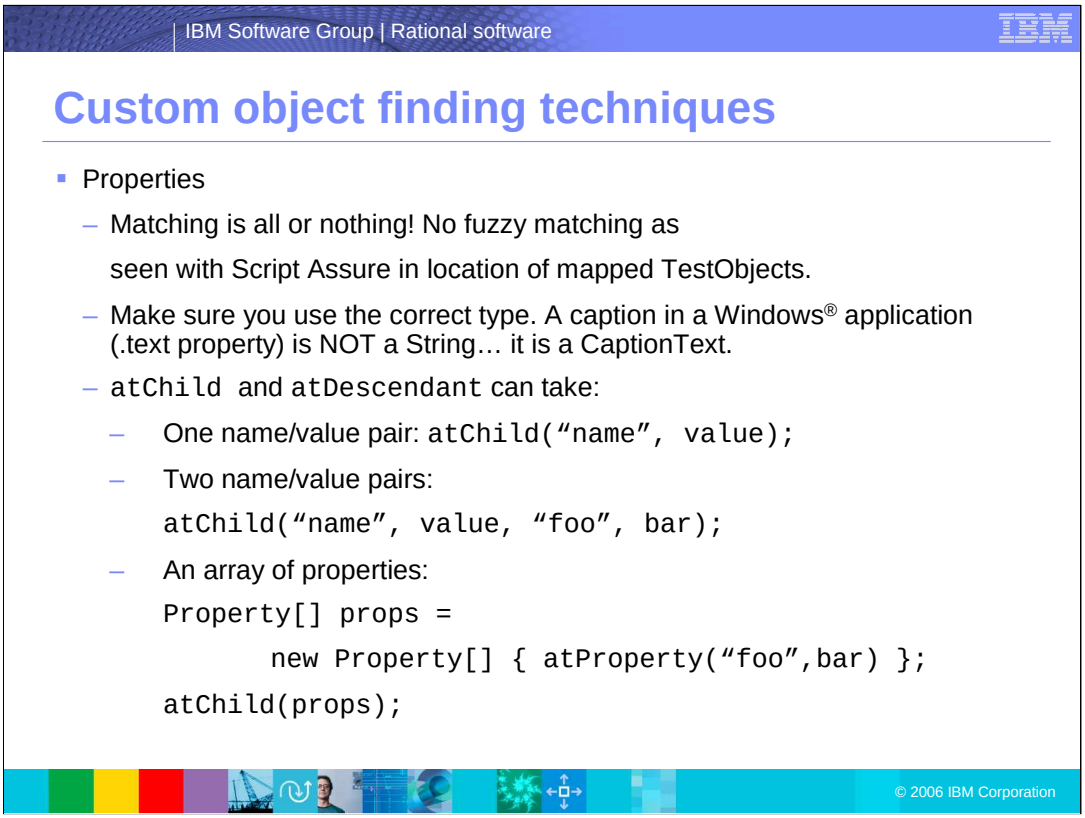
- Two main syntax concepts:
  - Relationship
    - Direct child
    - Descendant
  - Properties
    - Name/Value pairs
    - Same as properties in Object Map, `getProperties()`, and properties VPs.

There is a relationships concept and a properties concept. In the relationships concept, there could be a direct child relationship or a descendant relationship. Properties are the name/value pairs. The following slides will go into more details about each of these.

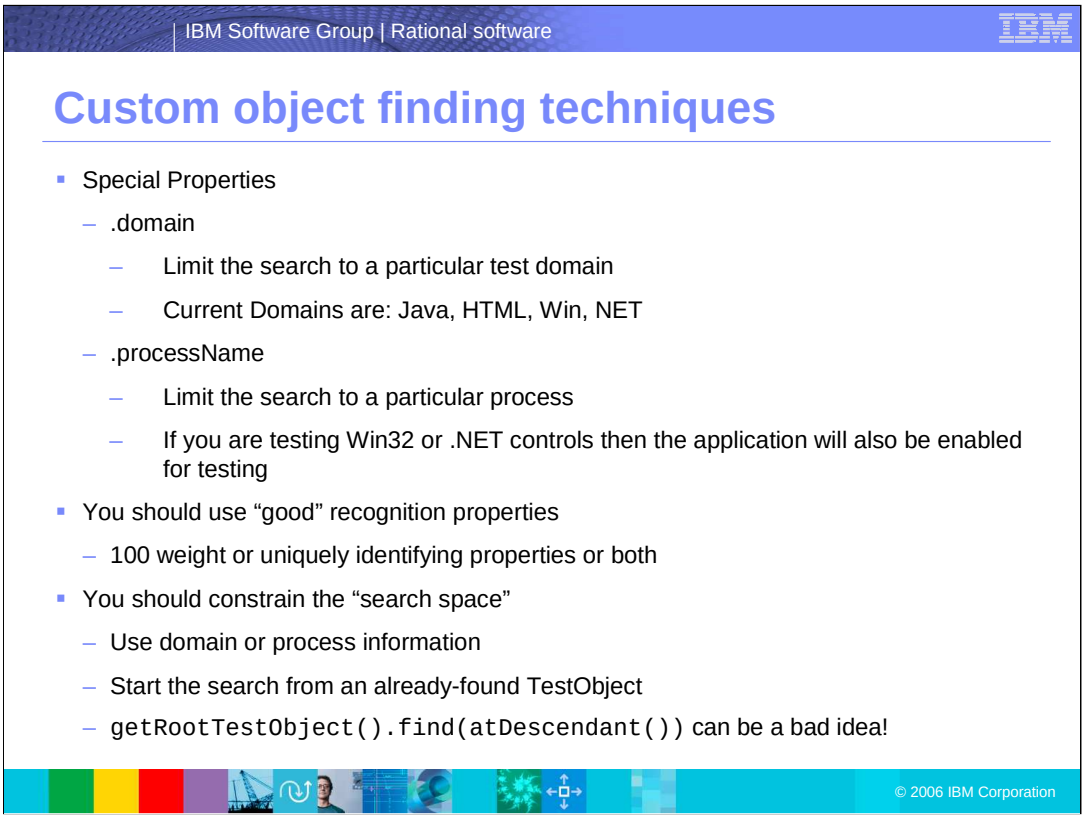
Custom object finding techniques

- Relationships
  - `atChild()`
    - Will only search direct children of the starting point
  - `atDescendant()`
    - Will search all descendants of the starting point
  - `atList()`
    - Used to link relationships together
    - Searches are performed in order
    - Express complicated relationships

In the relationship concept, the `atChild()` will only search direct children of the starting point, the `atDescendant()` will search all descendants of the starting point, and the `atList() will be` used to link relationships together. In addition, searches are performed in order, with each successive search starting from the result of the previous search, and can express complicated relationships like “find a dialog that is a child of a browser and get me any OK button in that dialog” in one API call.

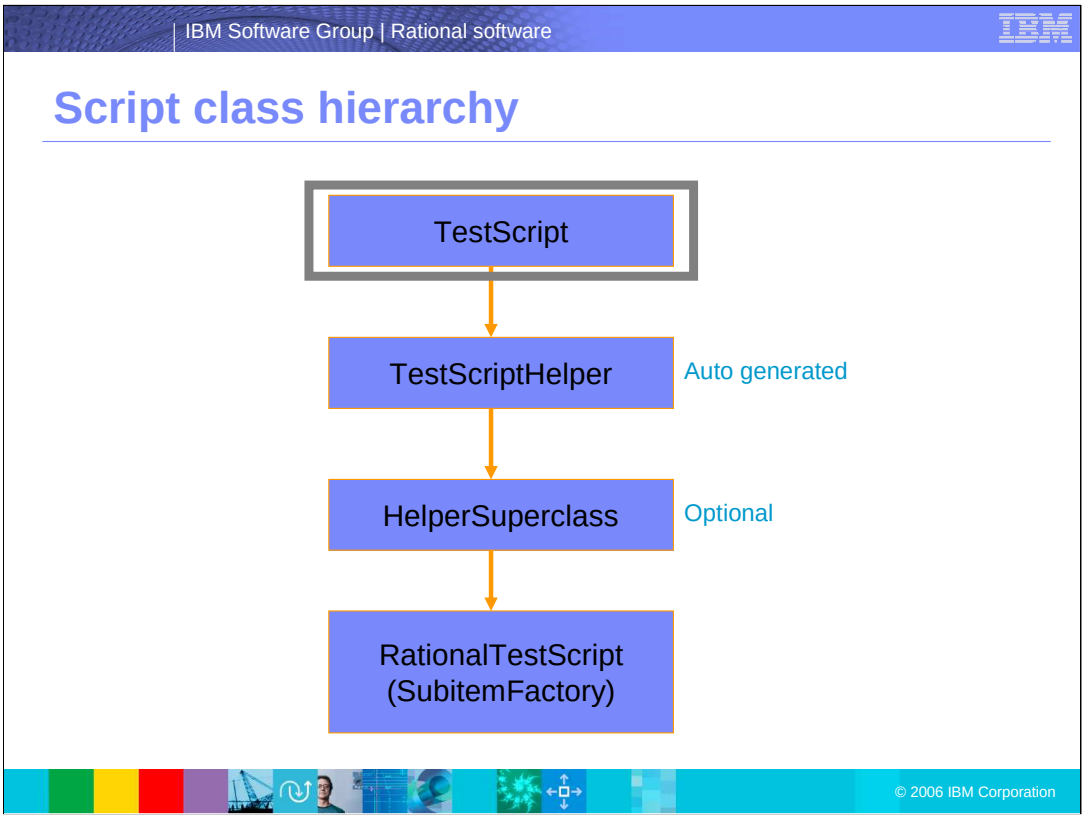


As for the properties, keep in mind that matching is all or nothing. Unlike ScriptAssure, you cannot use fuzzy matching. In addition, ensure you use the correct type as shown in the examples here.



Finally, there are special properties. The .domain and the .processName properties. The .domain property limits the search to a particular test domain. And the .processName property limits the search to a particular process.

You should use “good” recognition properties and constrain the “search space”

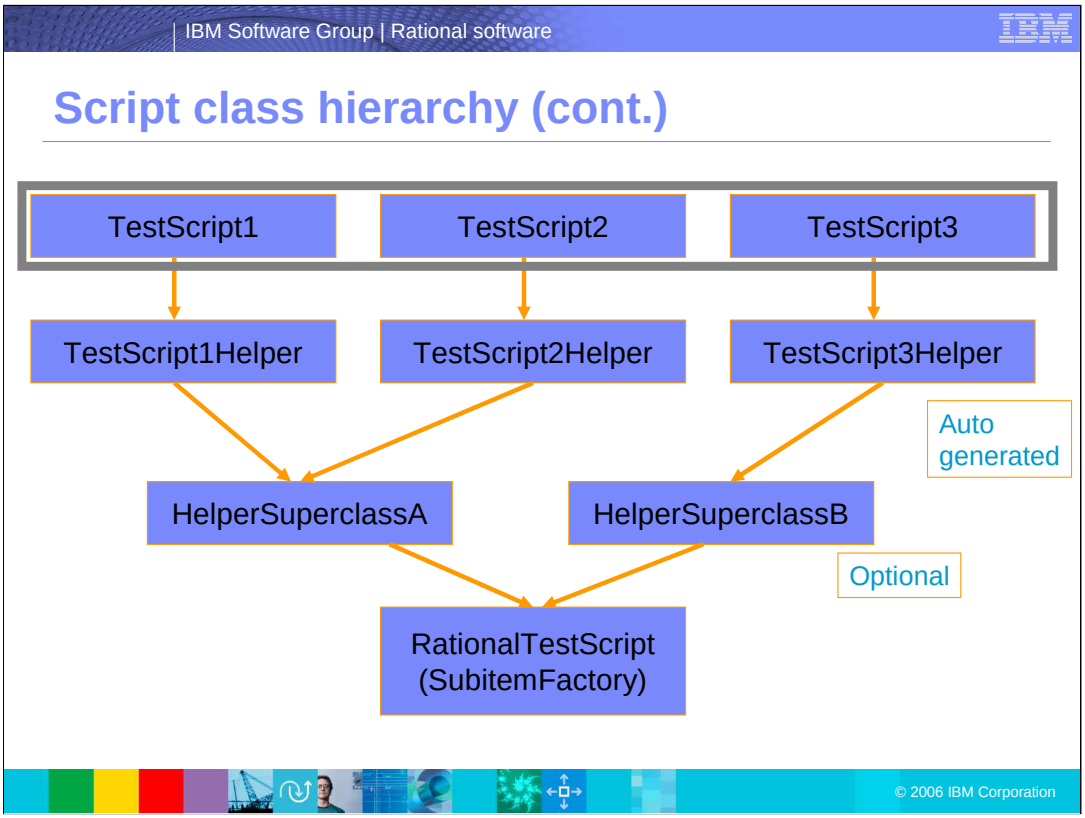


TestScript is the class that you primarily interact with.

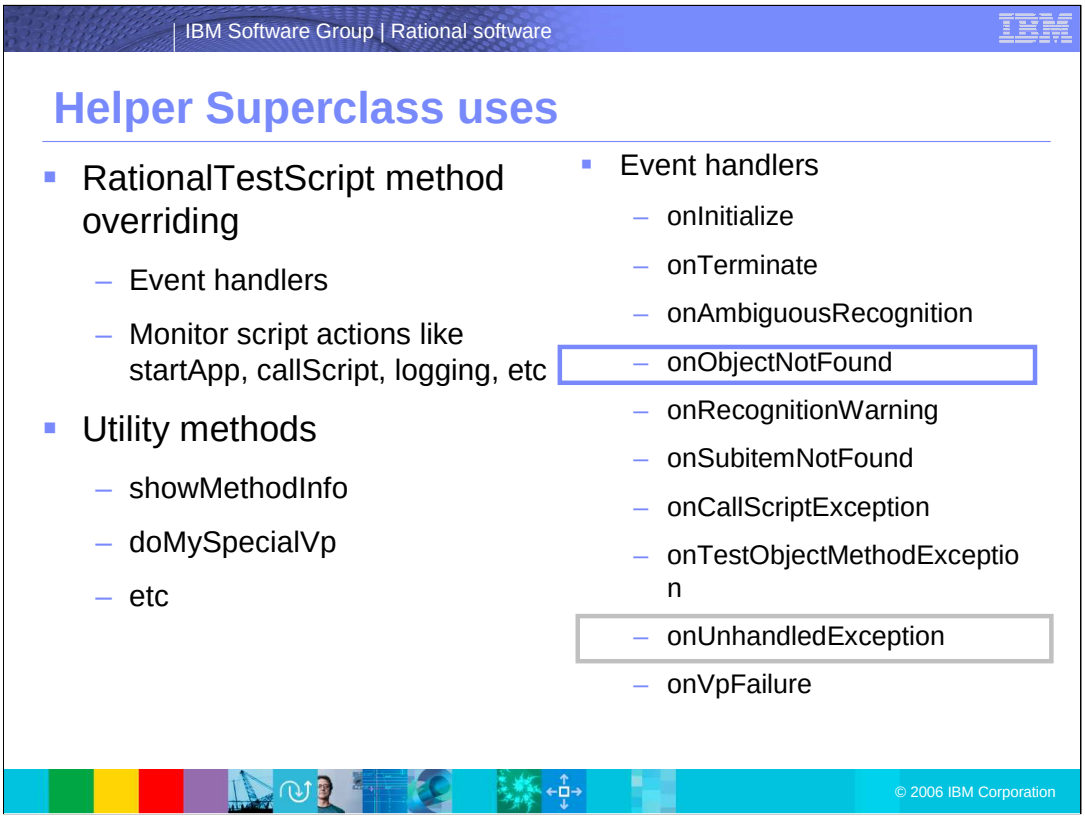
Helper is automatically generated by Rational Functional Tester based on Mapped Test Objects referenced by the script.

Helper super class is optional, added at creation if preference set, and is modifiable in script properties.

RationalTestScript is the base class for all scripts. The Helper super class MUST extend this class in the end and may have multiple levels in the helper super class inheritance hierarchy as long as RationalTestScript is at the root.



In an expanded hierarchy, the same inheritance hierarchy apply here. Multiple scripts can be extended using the same helper super class as shown here.



You can override the following methods in order to customize your application. There are also some utility methods you can invoke like the showMethodinfo and doMySpecialVP. Event handlers allow you to have your own implementation when certain events occur.

Advanced example: “Last Chance” find

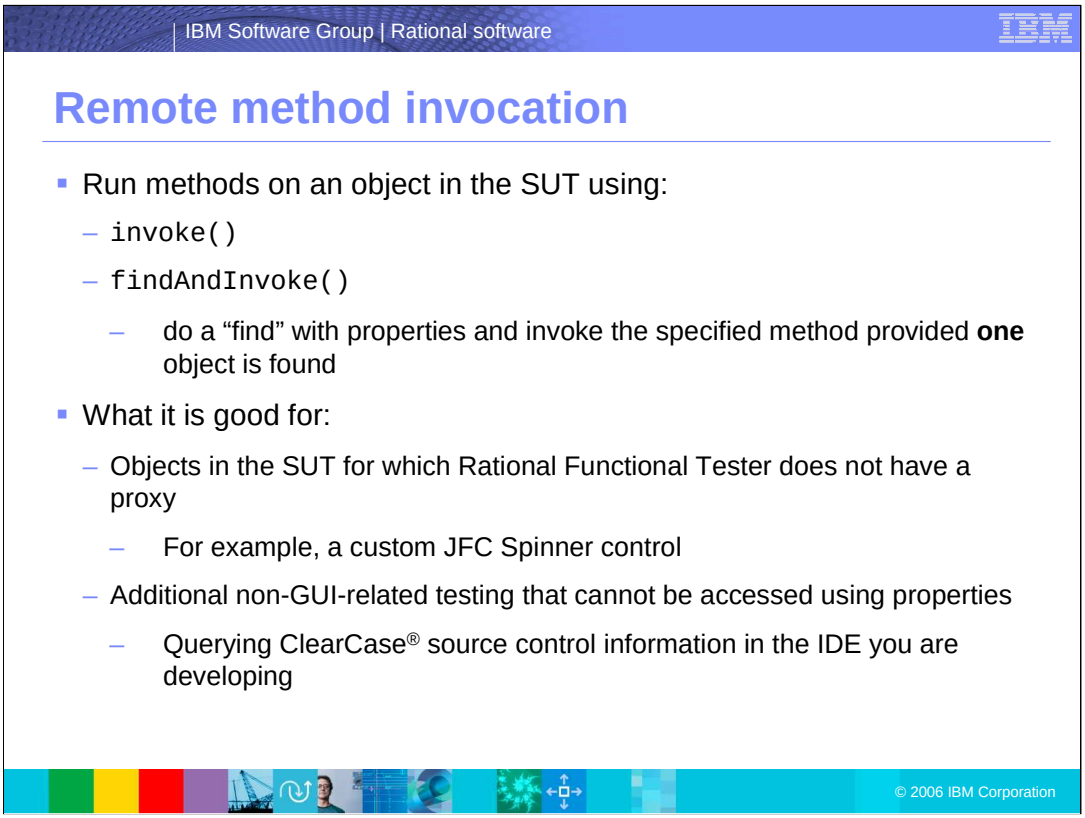
- The Problem:
  - The GUI hierarchy in the application you are testing sometimes changes
  - Script Assure requires rigid hierarchy
- Concepts Applied:
  - `find()` using properties
  - `OnObjectNotFound()` exception handler
  - Programmatic access of Mapped TestObjects
  - Using `TestObjectMethodState`

This advanced example explores the ‘last chance’ find. The problem: The GUI hierarchy in the application you are testing sometimes changes, and Script Assure requires rigid hierarchy.

Advanced example: “Last chance” find

- The Solution:
  - Override the `OnObjectNotFound()` exception handler
  - Gain access to the recognition properties of the Mapped TestObject you were trying to interact with
  - Use `find()` to try and locate the object without hierarchy constraints
    - If more than one result, throw an `AmbiguousRecognitionException` or pick one of the results (danger: “gliding”)
    - If no results found, continue with `ObjectNotFoundException`
    - If one result, set the found object in the `TestObjectMethodState`

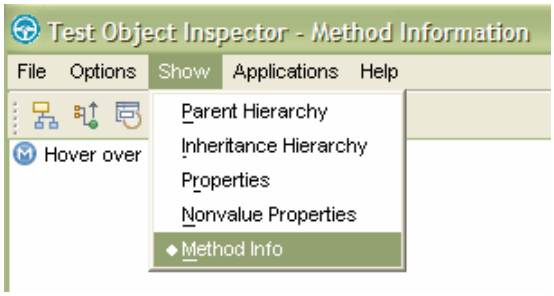
The solution: you should override the `OnObjectNotFound()` exception handler and gain access to the recognition properties of the Mapped TestObject you were trying to interact with. You can also use `find()` to try and locate the object without hierarchy constraints.



Remote Method Invocation is used by invoking the Software Under Test’s own method from Rational Functional Tester. It calls methods on an object in the Software Under Test through the invoke() and findAndInvoke() methods. Remote method invocation is beneficial in some scenarios such as when objects in the SUT for which Rational Functional Tester does not have a proxy and for non-GUI-related testing that cannot be accessed through the properties controls.

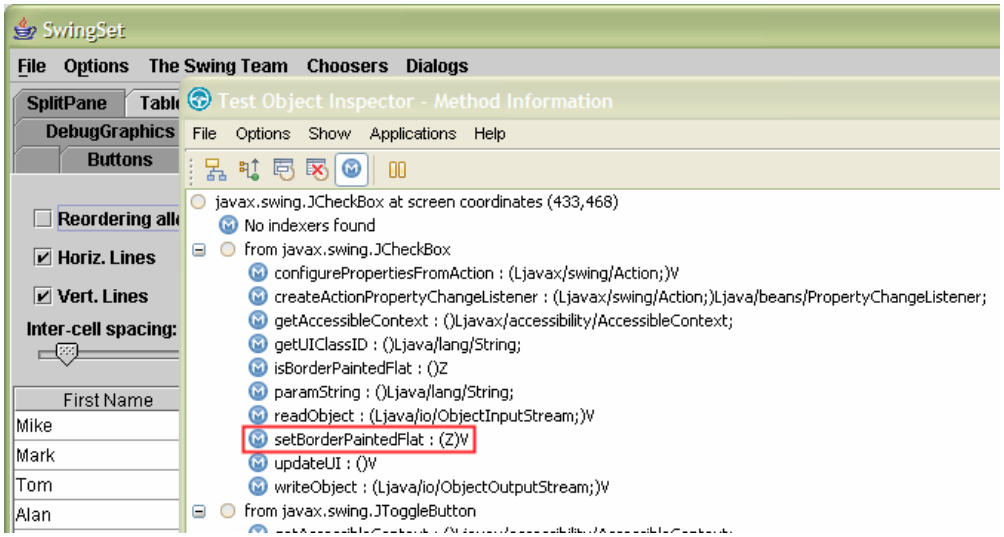
Remote method invocation

- When invoking methods that take arguments, the method signature needs to be passed to `invoke`.
  - The method signature uses JNI (Java Native Interface) syntax
  - The Test Object Inspector is your best friend!



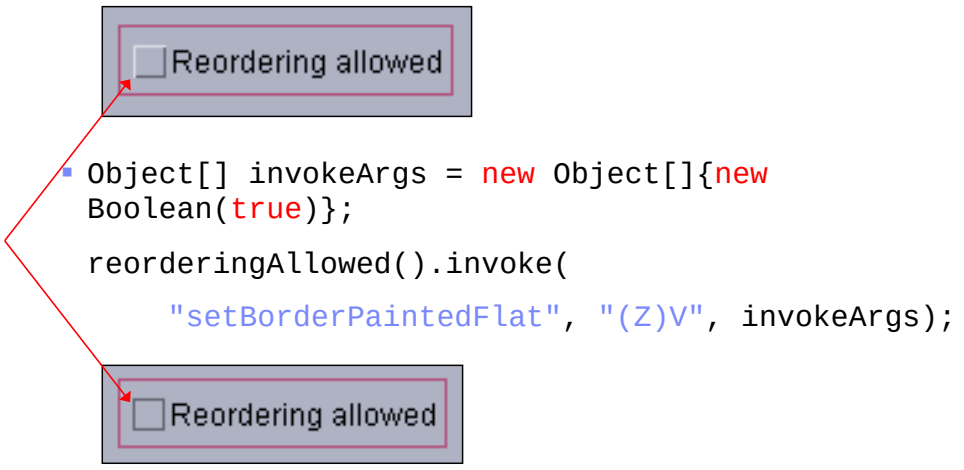
When invoking methods that take arguments, the method signature needs to be passed to `invoke`. The method signature uses Java Native Interface syntax. Use the Test Object Inspector to view method information as shown here.

Remote method invocation

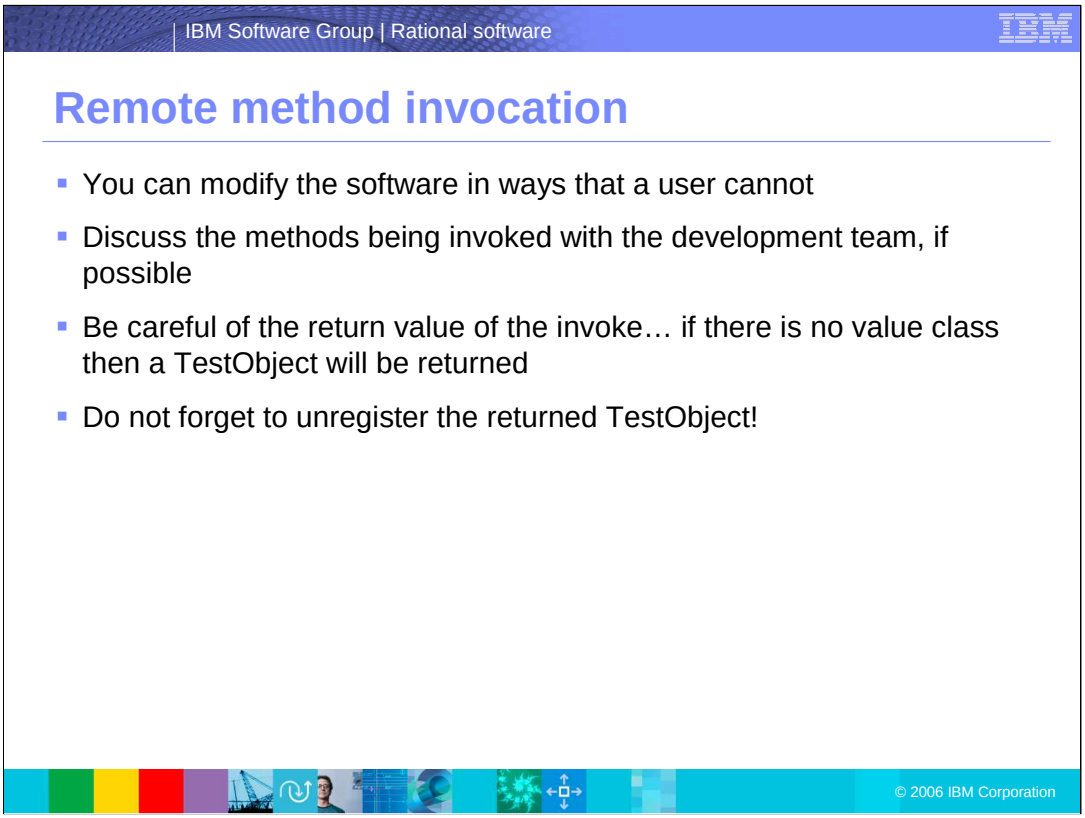


Shown here is an example of the list of the methods and JNI parameters from the Test Object Inspector.

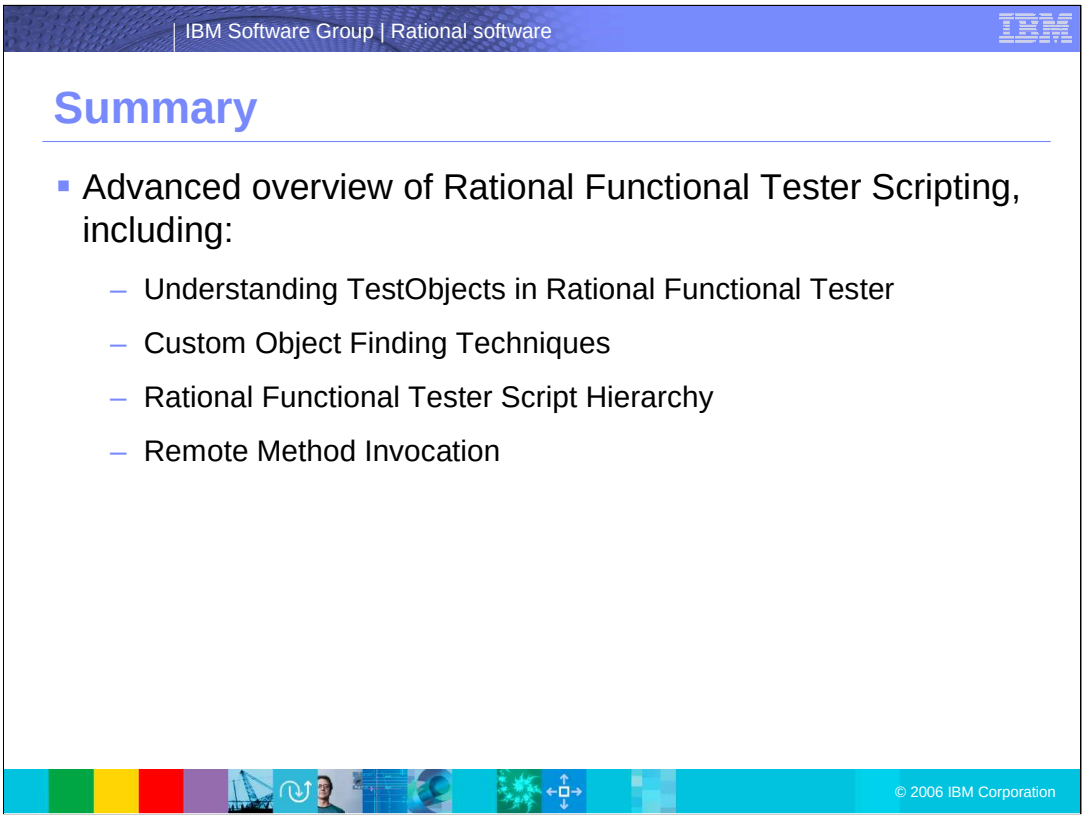
Remote method invocation



The example shown here shows you how to call the invoke() method with the correct parameters.



When using Remote Method Invocation, be aware that you can modify the software in ways that a user can not. You should discuss the methods being invoked with the development team. Be careful of the return value of the invoke… if there is no value class, then a TestObject will be returned, and finally, do not forget to unregister the returned TestObject!



In summary, you should now have a working knowledge of advanced scripting functions in Rational Functional Tester, including: TestObjects in Rational Functional Tester, Custom Object Finding Techniques, Rational Functional Tester Script Hierarchy, and Remote Method Invocation.

For more information on these topics, reference the Rational Functional Tester API on the online help.

Trademarks, copyrights and disclaimers

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM	CICS	IMS	MQSeries	Tivoli
IBM(logo)	Cloudscape	Informix	OS/390	WebSphere
e(logo)business	DB2	iSeries	OS/400	xSeries
AIX	DB2 Universal Database	Lotus	pSeries	zSeries

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are registered trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel, ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a registered trademark of Linus Torvalds.

Other company, product and service names may be trademarks or service marks of others.

Product data has been reviewed for accuracy as of the date of initial publication. Product data is subject to change without notice. This document could include technical inaccuracies or typographical errors. IBM may make improvements and/or changes in the product(s) and/or program(s) described herein at any time without notice. Any statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. References in this document to IBM products, programs, or services does not imply that IBM intends to make such products, programs or services available in all countries in which IBM operates or does business. Any reference to an IBM Program Product in this document is not intended to state or imply that only that program product may be used. Any functionally equivalent program, that does not infringe IBM's intellectual property rights, may be used instead.

Information is provided "AS IS" without warranty of any kind. THE INFORMATION PROVIDED IN THIS DOCUMENT IS DISTRIBUTED "AS IS" WITHOUT ANY WARRANTY, EITHER EXPRESS OR IMPLIED. IBM EXPRESSLY DISCLAIMS ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT. IBM shall have no responsibility to update this information. IBM products are warranted, if at all, according to the terms and conditions of the agreements (e.g., IBM Customer Agreement, Statement of Limited Warranty, International Program License Agreement, etc.) under which they are provided. Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products in connection with this publication and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. IBM makes no representations or warranties, express or implied, regarding non-IBM products and services.

The provision of the information contained herein is not intended to, and does not, grant any right or license under any IBM patents or copyrights. Inquiries regarding patent or copyright licenses should be made, in writing, to:
IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.