**Chapter 10. Advanced Scripting with Rational Functional Tester TestObjects**

**Jeffrey R. Bocarsly**

*Objects in a target application are represented in Rational Functional Tester scripts by objects derived from class* TestObject*.* TestObject*s are customarily used for two tasks in your Rational Functional Tester script implementation activities: manipulating Graphical User Interface (GUI) objects (administering clicks, performing data entry, and so on) and capturing data for verification against a baseline. This chapter examines some of the advanced features of the* TestObject *Application Programming Interface (API) that are relevant to these tasks. Because much of this discussion reflects aspects of how the Rational Functional Tester Object Map works, you might review* [*Chapter 9*](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09)*, “*[*Advanced Rational Functional Tester Object Map Topics*](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html#ch09)*,” before reading this chapter.*

*This chapter discusses five broad topics that should familiarize you with some of the layers underlying Rational Functional Tester’s* TestObjects *and the ways you can take advantage of* TestObjects *to extend the reach of your scripts. The first two topics address some of Rational Functional Tester’s basic design concepts underlying the* TestObject *classes, and some of the key requirements for using advanced* TestObject *technology. The third topic provides an extended discussion about how you can use key features of* TestObjects *to handle dynamic scripting situations where you need to exceed the constraints of the Rational Functional Tester Object Map. From there, the chapter moves on to the use of* TestObjects *to handle third-party controls. Finally, the discussion ends with a review of all of the different data-capture techniques available within Rational Functional Tester. Because these topics do build conceptually on one another, they are best read in the order in which they are presented.*

**Mapped TestObjects and Unmapped TestObjects**

The normal TestObject that you are used to working with during recording and script enhancement activities is a mapped TestObject. The term not only describes the fact that you access these TestObjects via the Object Map, but also serves to underscore the central concept of this chapter: in addition to referencing objects through the Object Map, Rational Functional Tester also allows you to reference *un*mapped objects. In other words, Rational Functional Tester gives you the option, through its API, of accessing objects completely *outside* of the Object Map.

Unmapped objects have a few different names in the Rational Functional Tester documentation—in addition to the term unmapped reference, they are also called bound or found references. This terminology refers to the fact that these references are TestObject references that are bound or found outside the Object Map apparatus. In permitting the user to go outside the Map apparatus, Rational Functional Tester provides you with tremendous flexibility, but it also places the burden on you to provide your scripts with what the Map provides under the hood: the setup for finding specific objects and the teardown to clean up after you are finished with your off-map TestObjects. For reasons that are mentioned in the next section in this chapter, the usual rule of letting garbage collection take care of your objects after you are done with them doesn’t apply when unmapped objects are used. When you go off-map, you are not only responsible for finding your objects, you also are responsible for cleaning up after you are done with those objects.

**Unregistering TestObjects**

Each time you create a *mapped* TestObject by calling a method from the Object Map, you invoke the standard Rational Functional Tester mechanism for attaching to an object in the target application and manipulating or interrogating it. Whenever you do this, Rational Functional Tester takes care of all setup and teardown activities, including disconnecting any TestObjects from their targets to let the remote Java Virtual Machine (JVM) or Common Language Runtime (CLR) release any memory resources via garbage collection that might be tied up by Rational Functional Tester. Although TestObjects are regular objects (in the JVM or CLR sense), they are unusual in that they can hold references to their target objects in the (remote) target application process. Because of this atypical arrangement, special care is required when destroying TestObjects, due to the following possibility: if Rational Functional Tester holds on to a reference to a target object when the target application has abandoned that object to garbage collection, *the garbage collector will not collect the object due to Rational Functional Tester’s reference to it*. Memory is not released, which ultimately can lead to unpredictable behavior. Under normal recording and playback circumstances, all the required cleanup activity is handled under the hood by Rational Functional Tester.

However, when you create your own unmapped references, Rational Functional Tester assumes that you are taking responsibility for all aspects of TestObject management, including cleanup activities. Termination of script execution in this case *does not* automatically result in cleanup of bound references. If you do not handle bound reference cleanup in your code, the situation described previously with Rational Functional Tester holding on to bound references to remote objects that are not garbage collected by their own virtual machines can occur. This problem happens with any number of bound references, but it can become especially pronounced if your script creates a large number of bound references that are not cleaned up. In this case, a large amount of memory might be involved, which results in resource constraints and the unpredictable behavior already mentioned. This is why bound references must be explicitly cleaned up, or *unregistered*.

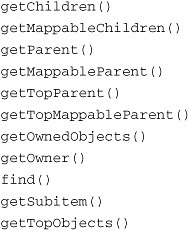
Bound references are unregistered by calling one of the unregister methods (obj.unregister(), unregister(Object[]), or unregisterAll()). These method calls mean that you can release bound references on an individual basis, in groups, or all at once. Individual bound references are released using the obj.unregister() method call, where obj is a bound reference. If you have an array of bound references, the unregister(Object[]) method is used; all current bound references can be released with unregisterAll(). It is also useful to note that the method getRegisteredTestObjects() returns an array of all currently bound TestObject references.

**Finding Objects Dynamically**

There are a number of reasons why you might want to discover objects dynamically. Typically, the need to do this occurs when target objects are in some way created on the fly, and there is no way to capture key recognition properties in the Object Map at design time, because they are not determined until run time. In this situation, a strategy that works well is to record into the Object Map a stable non-dynamic parent object, and then to use one of the methods in the Rational Functional Tester API to dynamically hook the interesting objects that have instance-based recognition properties.

**Obtaining Bound TestObject References**

The Rational Functional Tester API contains multiple methods that dynamically return bound references:



Before you look at these methods in detail, a few concepts need to be defined. On looking at this list, you can see that three of the methods contain the term Mappable. As noted in [Chapter 3](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch03.html#ch03), “[General Script Enhancements](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch03.html#ch03),” Rational Functional Tester has a concept of a mappable object, which does not mean that an object can or cannot be mapped; rather, a mappable object refers to an object that is *useful* to have mapped. In other words, objects that are not *interesting* from a Rational Functional Tester perspective are not considered mappable. Usually, objects are not mappable because they do not hold interesting application data or controls.

Another concept from the list of methods is that of the top object, or the top-level window. The top-level window of an application is the window at the root of the application window hierarchy. In the Windows operating system, top-level windows are windows that technically are not child windows of other windows but *are* parent windows of other windows (however top-level windows act in some ways as children of the desktop window). Each top-level window is the parent of its’ application window hierarchy.

A final concept that appears implicitly in the names of the methods in the list is the idea of owned and owner objects and parent and child objects. These two sets of terms reflect two parallel hierarchies that exist together in the Windows operating system, the owner/owned and the parent/child hierarchies. In the parent/child hierarchy, children are contained by parents and inherit from their parents. Windows that are owned are not constrained by their owners in this way—ownership is typically a relationship *between* top-level windows (for example, usually a dialog is owned by a top-level window, but is not its child; both the dialog and its owner are top-level windows). Owned windows, however, have to follow certain state rules that rely on the state of their owners.

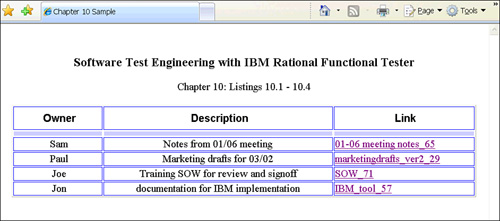
With these concepts, it becomes easy to understand what most of the methods in our list do. For example, getChildren() returns an array of TestObjects representing all of an object’s children, whereas getMappableChildren() children returns a TestObject array of just the mappable children. A similar relationship exists for the pair getParent() and getMappableParent() and the pair getTopParent() and getTopMappableParent(). The methods getOwner() and getOwnedObjects() likewise return TestObjects representing an object’s owner and owned objects.

All the methods discussed are methods of the TestObject class, and calls into these methods are straightforward: none of them takes an argument, and they return a TestObject reference or an array of TestObject references.

The remaining methods in the “bound reference” list are a bit special, and they require a more detailed examination. The first of these is the find() method, which has a couple of overloaded versions that make it a powerful tool for obtaining references to TestObjects. In its simplest form, you can call find() with no arguments on any TestObject, and it returns a bound reference to that object. However, if you provide find() with a Subitem argument defining search criteria, it acts as a powerful search engine to *find* objects that satisfy your search criteria.

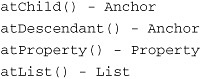
To see how this version of find() works, consider the problem of a table in a web page that contains variable links—imagine the links represent files that are uploaded by users for other users to download. [Figure 10.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10fig01) shows a sample page that can be used to illustrate this—the links change every time the page is loaded (the page is included in the Supplementary material for the text at [www.ibm.com/developerworks/rational/library/09/testengineeringrft/index.html](http://www.ibm.com/developerworks/rational/library/09/testengineeringrft/index.html)).

**Figure 10.1** A sample web page with variable links



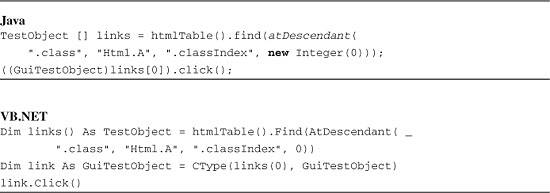
Because the names of the links are unpredictable, you have to search for the links that are children of the HTML table in order to click on a specific link. To construct your search, find() takes a Subitem argument that specifies the search criteria. There are 16 different types that inherit from Subitem: Anchor, Button, Cell, Column, Date, Id, Index, List, Location, Position, Property, Row, Separator, Value, Week, and Weekday.

However, find() executes with only three of them (Anchor, Property, and List); otherwise, it throws an InvalidSubitemException exception. For find(), the relevant SubitemFactory methods (with the types of their return values) are:



You might construct a search with the find() method using two properties, the. "class" and the ."classIndex" properties, as shown in [Listing 10.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex01).

**Listing 10.1** A first call to find()



In [Listing 10.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex01), find() searches for a “.class” property of value Html.A and a “.classIndex” of 0. Note that the numeric is passed as an Integer. The exact same effect can be achieved with a Property object, as shown in [Listing 10.2](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex02).

**Listing 10.2** Calling **find()** with a **Property** object



This second syntax using a Property object enables you to define as many properties as you would like to use as constraints for your search. The first syntax relies on overloaded versions of atDescendant(), which takes at most two pairs of property arguments.

Note

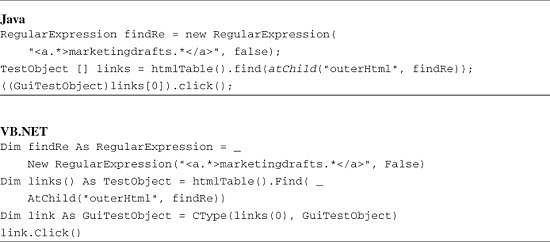
In the VB.NET listing ([Listing 10.2](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex02)), the fully qualified name for the Rational **Property** class is required, as .NET has a **Property** namespace, and the compiler cannot distinguish between the two, even with an explicit **Imports** statement.

It is worth noting that there is a significant difference between passing a pair of property-value arguments to atDescendant() (as in [Listing 10.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex01)) and passing a list of atDescendant() arguments using atList() (shown in the following code snippet). In the snippet below, find() does *not* find *any* target objects, because this search is looking for a link descendant of the HTML table that *itself* has a descendant with a .classIndex property of 0. In [Listing 10.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex01), the search is looking for a descendant of the HTML table with both a .class property of value Html.A *and* a .classIndex property of value 0.

image

There is another dimension to the use of find() that makes it even more powerful than what you have already seen. This is the ability to specify the target value for a property not as an actual value, but rather as a regular expression. To use this functionality, the regular expression pattern is defined using the RegularExpression class and is then passed in to the Subitem call as an argument. For example, if you want to find the link or links that contain the phrase “marketing-drafts” in the link name, you can call find() with a RegularExpression argument as shown in [Listing 10.3](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex03).

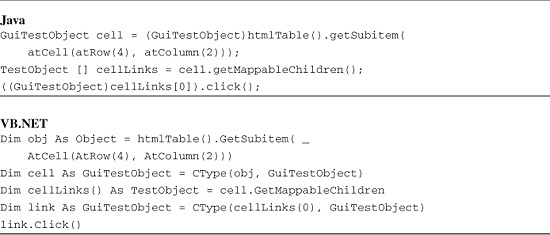
**Listing 10.3** Calling **find()** with a **RegularExpression** argument



As you can see, the RegularExpression constructor takes two arguments in this example. The first is the regular expression pattern, and the second is a boolean specifying whether the comparison is to be case sensitive or not. In [Listing 10.3](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex03), the comparison is not case sensitive. This enables you to use a simpler regular expression than otherwise; for example, you don’t have to specify in your regular expression whether to accept tag names that are both capitalized and not capitalized with the second argument set to false. If you want to control every aspect of the matching from your regular expression, just set this argument to true. Finally, as a side note, although the RegularExpression constructor used here is the one you are most likely to use, it is overloaded—see the RegularExpression class documentation for the other versions (Rational Functional Tester Help menu > Functional Test API Reference).

Rational Functional Tester provides another route to locating objects on the fly, and this is the method getSubitem(). getSubitem() is found in all classes implementing the IGraphicalSubitem interface (StatelessGuiSubitemTestObject and its descendants) and a few of the classes that inherit from Subitem. The method takes Subitem arguments that define a location in an object. Usually, the objects this approach is appropriate for are data-rich objects such as menus, lists, and tables. A menu path is defined using the method atPath(), whereas a list position is defined using atIndex(), and a cell location on a table is identified using atCell() in conjunction with atRow() and atColumn(). With these methods, getSubitem() can be used to solve the same problem solved in [Listing 10.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex01) and [Listing 10.2](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex02) with find()—how to click on the link in an HTML table cell without knowing the actual link text at design time. [Listing 10.4](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex04) shows the code.

**Listing 10.4** Using getSubitem()



[Listing 10.4](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex04) shows how you can define the cell that you’re interested in by its coordinates (4, 2), and then query for its mappable children. Assuming that the only mappable child is its link child, you can use standard Rational Functional Tester code to click on the target link. If there are multiple mappable children, you might want to combine the getSubitem() approach with a find() on the resulting cell to filter out the cell children that are uninteresting.

There is one final method from the list of methods that return bound references for you to review. This is the method getTopObjects(). This method returns all the top-level objects (windows) in a domain that Rational Functional Tester supports, as an array of TestObjects. The ability to access top-level objects in a domain gives the Rational Functional Tester user a way to access all the objects in that domain without recourse to the Rational Functional Tester Object Map. This method is discussed in the next section.

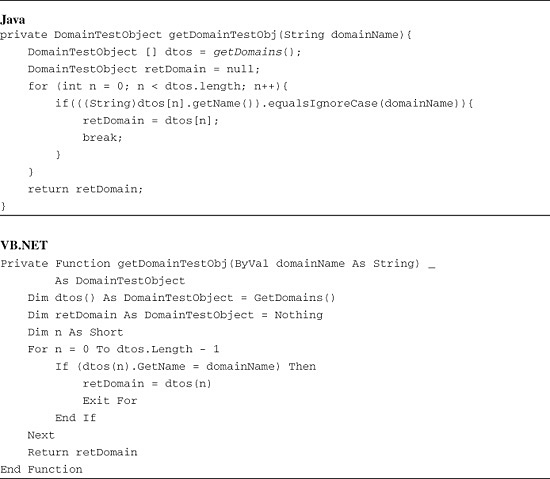
**Living Off the Map**

Rational Functional Tester, in addition to all the advanced functionality of its Object Map, provides you with the ability to go off-Map completely. This amounts to building your own map, but it certainly is a good option to have, and in some cases, the amount of extra effort required to build an implementation is justified.

The process starts by obtaining a set of objects that represent the running Rational Functional Tester domains. These are a set of special TestObjects called DomainTestObjects. The method getDomains() returns an array of DomainTestObjects that represent all domains that are currently running. Note that there can be more domains running than you are aware of, because, for example, Rational Functional Tester connects to the Internet Explorer browser through its Java plugin, so when you are working in the HTML domain with Internet Explorer, both the HTML and the Java domains are in use. In addition, Rational Functional Tester creates DomainTestObjects for its own purposes, so there are usually DomainTestObjects running that you are not actually using.

You can identify the specific DomainTestObject that you need by calling the method getName() on each DomainTestObject and selecting based on the name. For example, you select the Html domain for a web application, the Java domain for a Swing application, the Net domain for a .NET application, and so on. [Listing 10.5](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex05) shows a method that implements a simple domain search.

**Listing 10.5** Getting a domain’s DomainTestObject



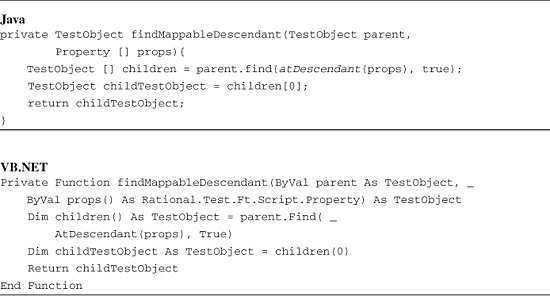
After you have the correct DomainTestObject, you can call getTopObjects(), and this supplies you with an array of TestObjects for all the top-level windows in the domain. When you have your domain’s top-level objects, you are living off the Map. [Listing 10.6](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex06) illustrates how simple this is with the method getDomainTestObj() from [Listing 10.5](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex05).

**Listing 10.6** Getting the top-level windows of a domain



With the top-level windows in a domain in hand, you are ready to search the object tree underneath the top-level windows for specific objects. To illustrate, consider the same problem that was solved previously—finding a link of unknown text in an HTML table. The top-level window in this case is the browser, and one way to capture an element of a page would be to use find() on the top-level window to obtain a TestObject reference to one of its descendants. [Listing 10.7](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex07) shows how this can be done.

**Listing 10.7** Getting descendants of a top-level window

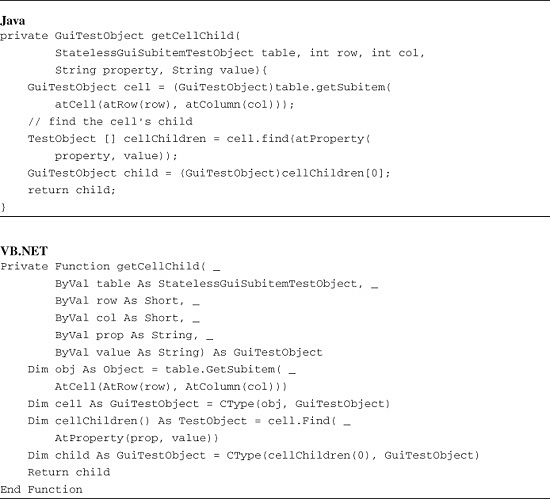


A few comments about the findMappableDescendant() method in [Listing 10.7](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex07). In this method, the search criteria are assumed to be specific so that only one child is found, and a reference to that child is returned. That might not always be true, and you might want to extend your own code to include other possibilities. Second, the search criteria are defined by an array of Property references instead of a single property-value pair, so that you can use multiple criteria. Finally, the code is set up to search only for descendants of our parent argument, which is the broadest search—using atChild() limits the search to immediate children of the parent object and would execute more quickly.

It is worthwhile to reflect for a moment on the use of find(). As noted, find() is a powerful tool, but on complex web pages or forms, invoking find() (especially with an atDescendant() argument) might mean that you perform a full (recursive) search of the object tree every time you call your method. That can involve a serious amount of overhead, especially if you use a high-level parent object relative to your actual search target. Performing a large search every time you execute can make your scripts run slowly. If you plan to shape your solution this way, test the implications of your approach carefully before deploying your test suite.

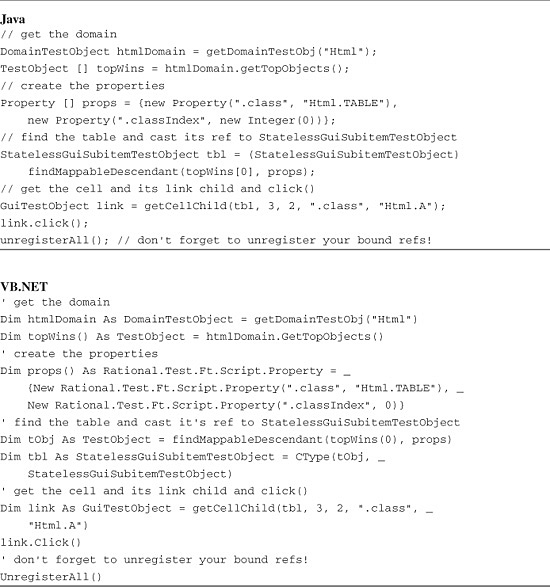
You can use the method findMappableDescendant() to find a specific HTML table (for example, you can uniquely identify an HTML table in a page by its .classIndex property). Once you have the table reference you need, you can identify a specific cell using getSubitem() and hook its link (a child of the cell) with find(). This is shown in [Listing 10.8](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex08).

**Listing 10.8** Getting a table cell and its children



The methods assembled provide a simple, live Map implementation, which you can use to build a script for your application. Sample code to find a specific link by cell coordinates (row 3, column 2) and click on the link is shown in [Listing 10.9](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex09). [Listing 10.9](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex09) starts with the code from [Listing 10.6](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex06) and calls the methods developed in [Listings 10.7](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex07) and [10.8](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex08).

**Listing 10.9** A sample off-map script



As you can see, the methods in the Rational Functional Tester API that make it possible for you to search for TestObjects on the fly give you a powerful set of tools for dynamically working with objects in your target application. As a final (cautionary) thought about dynamically tracking TestObjects, before you spend time and effort building a full custom implementation, consider whether you can use the Rational Functional Tester Object Map for your needs. The Rational Functional Tester Object Map with its feature set is a robust, flexible solution to the problem of tracking objects, and you might be able to meet your needs with it, rather than expending the effort of building your own custom framework.

Note

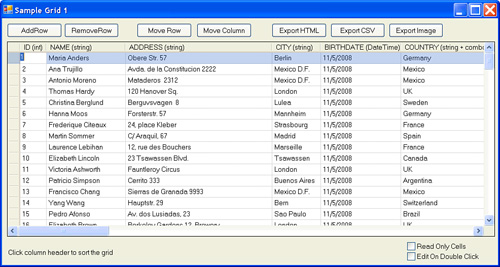
Remember, all the TestObject references used in this discussion are bound references; therefore, you will need to unregister each of them with one of the unregister methods at the appropriate place in your code. Typically, this can be done at the end of a script; however, if your code generates a large number of bound references relative to the amount of memory you have available, you might have to be more aggressive about managing your bound references.

**Handling Third-Party Controls**

Third-party controls have long been a challenge for automated testing. This is because although the tool vendors have always committed to supporting the standard controls supplied by the IDE vendors and perhaps a selection of the most popular third-party controls, there are always more third-party control packages available than can be conveniently supported by the tool vendors. If your project uses an unsupported third-party control package, you have often been out of luck. With Rational Functional Tester, you are frequently *in luck* because Rational Functional Tester offers you an API to handle third-party controls.

To illustrate how to handle a third-party control with Rational Functional Tester, the following discussion uses an open-source .NET control. This is the DevAge SourceGrid control (copyright [www.devage.com](http://www.devage.com/)), which is a flexible and powerful .NET winforms grid control written in C#. The examples that follow use SourceGrid version 4.11 with the DevAge reference application (you can download the reference application with the DevAge package and run the examples on it without having to write your own application). One presentation of the SourceGrid control is shown in [Figure 10.2](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10fig02) (this is Sample Grid 1 in the reference application); the following example focuses on this grid.

**Figure 10.2** The DevAge SourceGrid control



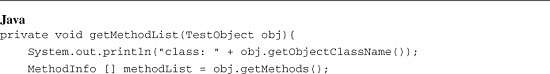
**The General Strategy**

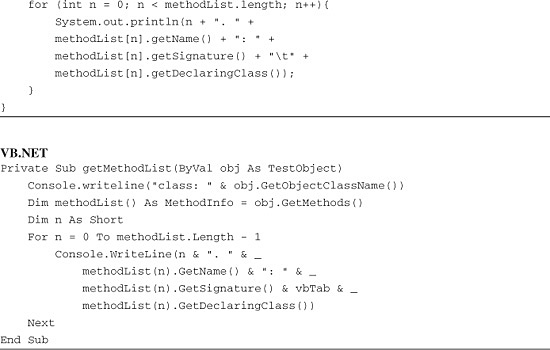
When you handle a third-party control that is unhandled by the standard Rational Functional Tester mechanisms, the general approach is to discover the native methods and properties of the unhandled control with the Rational Functional Tester API to learn how to either manipulate the control for testing purposes (for example, create a click event on the control) or to capture control data. Similar to other tools in its class, Rational Functional Tester gives you direct access to the control object and its children. Rational Functional Tester does this not by giving you a live reference to the control in your script, but rather by giving you a bound TestObject reference to the control, just as you saw in the previous section. You can use these bound references to call the native methods and properties of the target control.

**Scoping Out the Third-Party Object Hierarchy**

The first task with any third-party control is to scope out the object hierarchy, so that you have the information you need about the control to use a bound reference to it. You need to understand what the relationships are between the objects from which the control is composed and the methods on each object that are available for you to call. The simplest way to start doing this is with Rational Functional Tester’s Test Object Inspector tool. The Test Object Inspector tool gives a fast and direct way to get a complete picture of object methods and properties. The Test Object Inspector has a significant limitation, however, and that is that it only gives you information about specific objects in the hierarchy. Objects that are deeper in the hierarchy are not visible with the Inspector, so you have to explore the object tree by hand. You can explore with a few simple tools built from the Rational Functional Tester API. The key API method to start with is the TestObject method getMethods(), which returns an array of MethodInfo references that describe the methods each object exposes. You can use getMethods() to build a simple method-spy method, which is shown in [Listing 10.10](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex10).

**Listing 10.10** A spy method to spy methods on unhandled objects



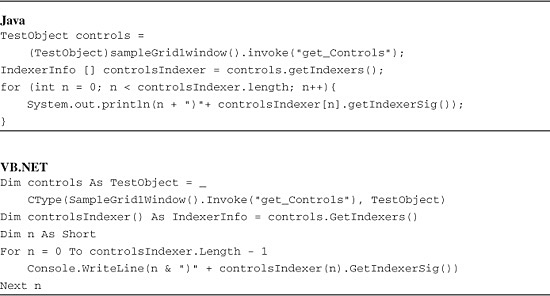


The getMethodList() method in [Listing 10.10](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex10) prints each method, its signature, and its class to the Eclipse console. You can use this list to find methods that are likely to help you drill down into the control to obtain references to interesting and useful child objects.

To make this process more concrete, the following discussion works through an example. The example comes from the DevAge SourceGrid ([Figure 10.2](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10fig02)). The process starts by exploring what grid objects can be captured into the Rational Functional Tester Object Map. This turns out to be only the winform container of the grid control. If you use getMethodList() and pass in a reference to the winform object in the object map, 1,028 methods are returned for the form. Several methods look like they might provide access to the controls on the form (for example, get\_ActiveControl, get\_TopLevelControl, and get\_Children), but the most promising is get\_Controls, which returns a reference to the form’s controls collection.

After you know the name of the method you’re interested in calling, the mechanism by which you get access to target object methods is by calling the TestObject method invoke(). The invoke() method is overloaded, but this example requires only the simplest version, in which invoke() takes a single argument: the name of the method you want to call. When you pass in get\_Controls as the argument, invoke() returns a TestObject that wraps the form’s controls collection. Because the controls collection is a collection object, you can expect that it has methods to return references to individual members of the collection. In Rational Functional Tester parlance, this is called an indexer, and you can use the method getIndexers() on your TestObject reference (which returns an array of IndexerInfo references) to get information about the indexers that are available on the collection. [Listing 10.11](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex11) illustrates a call to invoke() to get a control collection from the grid’s form and a query of the returned collection reference for indexers.

**Listing 10.11** A simple invoke() example



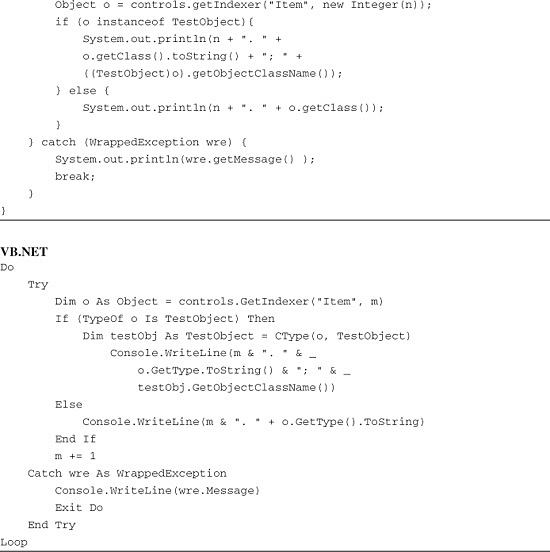
In [Listing 10.11](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex11), the mapped TestObject for the application winform (returned by sampleGrid1window()) is used to invoke the form’s get\_Controls method. The code then gets a list of the control collection’s indexers by calling getIndexers() on the returned TestObject reference, and each indexer method’s signature is printed to the console—indexers are really just enumeration methods. The following indexer information is printed by this code:

0) Item(I)  
1) Item(LSystem.String;)

This output tells you that there are two overloaded indexer methods to call—one takes an Integer argument, and the other a String (the notation used is discussed in the “[JNI Signature Types](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10sb01)” sidebar). The first signature offers the convenience of iterating through the collection using an index. By iterating through the collection the collection members can be interrogated for their class names, and if they are TestObjects, the class that each TestObject wraps. [Listing 10.12](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex12) shows this approach.

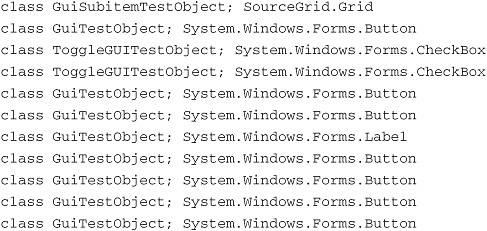
**Listing 10.12** Interrogating a collection object with an indexer

image



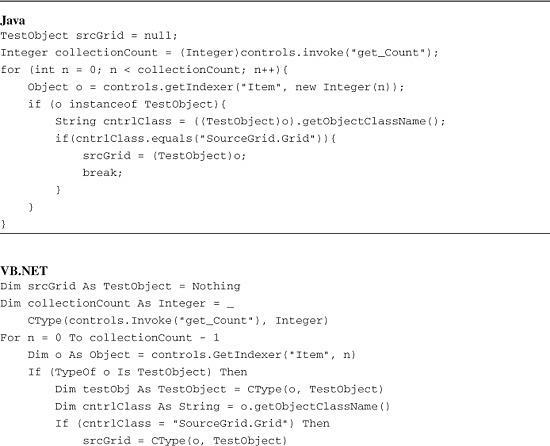
[Listing 10.12](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex12) has a couple of interesting features. First, indexers do not broadcast the size of their collection, so the size is unknown at the outset. This is handled in [Listing 10.12](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex12) by catching a WrappedException. This is the second interesting feature in [Listing 10.12](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex12)—what is wrapped by Rational Functional Tester’s WrappedException class? This is typically an exception class on the *target*. In this case, it is the .NET exception System.ArgumentOutOfRangeException. As you can see from the code in [Listing 10.12](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex12), this exception is caught regardless of whether you use the Java flavor or the VB.NET flavor of Rational Functional Tester.

The output from the code in [Listing 10.12](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex12) is rewarding because it shows you just the information that is needed (the output is abbreviated slightly to make it more readable):



From this output, you can see the TestObject types that wrap references to the target application GUI objects. In addition, you can see by class exactly what the target objects are, and this list tells you that the first entry is what you want—a reference to the SourceGrid control. With this information, it is straightforward to write some code to pull a reference out of the control collection corresponding to the SourceGrid, as shown in [Listing 10.13](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex13).

**Listing 10.13** Using an indexer to search a collection





[Listing 10.13](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex13) uses an approach that is similar to that in [Listing 10.12](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex12). However, in [Listing 10.13](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex13), the get\_Count method is invoked on the controls collection object to retrieve the number of control references in the collection. This removes the requirement for any exception handling inside the loop because the exact number of members is known.

It is also worthwhile to note that [Listings 10.11](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex11)–[10.13](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex13) are at least partly for purposes of illustration of how to cull through an object tree. You could accomplish the same thing using a call to find() on your mapped TestObject:

TestObject [] srcGrid = sampleGrid1window().find(  
         *atProperty*(".class", "SourceGrid.Grid"), false);

However, if you use this approach, you bear the cost of calling find() every time you run your script. With the code shown in [Listings 10.11](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex11)–[10.13](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex13), you need only to search through the controls collection, which might be a much more limited search, depending on where your target object is in the overall control hierarchy.

With the grid reference obtained in [Listing 10.13](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex13), you are now ready to see how to capture grid data and otherwise manipulate the grid via a bound reference.

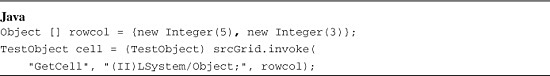
**Capturing SourceGrid Cell Data**

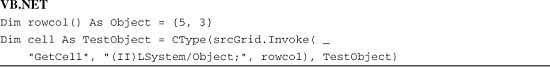
Now that you have a bound reference to the SourceGrid, you can use it to explore the grid control and how to capture data from the control. If you use getMethodList() on the grid, you find among the 970 grid methods a method called GetCell. This sounds just like what you want in order to get a bound reference to a cell object from the grid control.

To call GetCell, you have to learn about an overloaded invoke() method call, which takes not one but three arguments. The first argument of the three is the same as the argument to the one-argument version of invoke()—the remote method name. The second argument is a description of the remote method signature (see the “[JNI Signature Types](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10sb01)” sidebar), and the final argument is an Object array of arguments to pass to the remote method call.

The remote method call GetCell has the signature “(II)LSourceGrid.Cells.ICellVirtual;” according to the output from getMethodList(). This means that the remote method call takes two int arguments and returns a reference to the SourceGrid type SourceGrid.Cells.ICellVirtual. [Listing 10.14](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex14) shows how to set up this call, using an arbitrary cell (the 5, 3 cell of the grid) for purposes of illustration.

**Listing 10.14** Three-argument **invoke()**



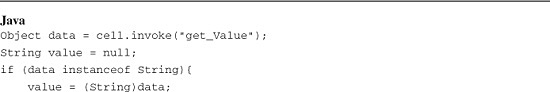


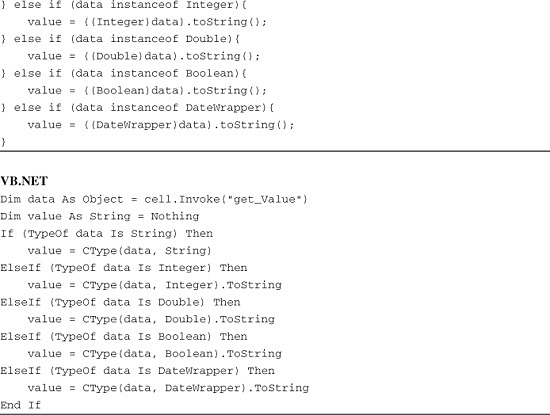
The first argument to invoke() is just the name of the remote method call, as you saw previously. The second argument is more subtle. According to the output from getMethodList(), this argument should be: “(II)LSourceGrid/Cells/ICellVirtual;”. However, with reference types in the return value signature, Rational Functional Tester might throw an InvalidSignatureException exception if it doesn’t have a reference to the type (even where the signature is clearly correct based on the getMethodList() output). This is true even if you use the .NET flavor of Rational Functional Tester for a .NET application because the target application has referenced the type, not Rational Functional Tester. There are two solutions to this problem: one is simply to omit the return type from the signature (which is legal), and the other is to use a completely general typing—“LSystem/Object;” for this case. Both appear to work equally well. This gives the full signature “(II)LSystem/Object;” for the call to GetCell (or more simply: “(II)” without the return type) indicating that the method takes two int arguments and returns an object reference.

The final argument to invoke() is an Object array containing the values to be passed as arguments to the remote call. [Listing 10.14](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex14) uses the arbitrarily chosen 5, 3 cell in the grid, employing an array of type Object containing the Integers 5 and 3 to specify the cell to the remote call.

Now that you have a TestObject reference to a grid cell ([Listing 10.14](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex14)), you can perform the final step in data capture from the grid. If you run getMethodList() on a cell object containing text, the output shows that the cell class is SourceGrid.Cells.Cell and has 42 methods. One of those methods is get\_Value a good candidate method for retrieving the text data from a cell. If you hunt around the grid and check cells in different columns (by invoking get\_Value and calling getClass().toString() on the returned type), you see that invoke() returns a variety of different data types (all value classes): String, Integer, Double, Boolean, and DateWrapper, depending on the data type of the cell you hooked. To handle these different data types, we write a small filter to check for each data type and cast it to a String, because String is frequently the most useful form to evaluate data with. [Listing 10.15](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex15) shows the call to get\_Value and the handling for the return value.

**Listing 10.15** Getting values from cells





A couple of aspects of the code in [Listing 10.15](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex15) are worth noting. The data type returned depends on the data that the cell holds, so you need to handle whatever data types your grid might contain. The SourceGrid example has cells that contain the standard numerical types Integer and Double, along with cells that contain String data. In addition, the grid contains cells that have checkboxes, and the data from these cells is returned as Boolean data. Rational Functional Tester considers these value classes. One of the types of data returned by the SourceGrid does not appear to be a standard value class, and this is the DateWrapper type. In this case, Rational Functional Tester provides its own custom wrapper of a “standard” value class in order to add additional functionality. DateWrapper overrides some of the Date methods to provide more powerful ways to compare date values.

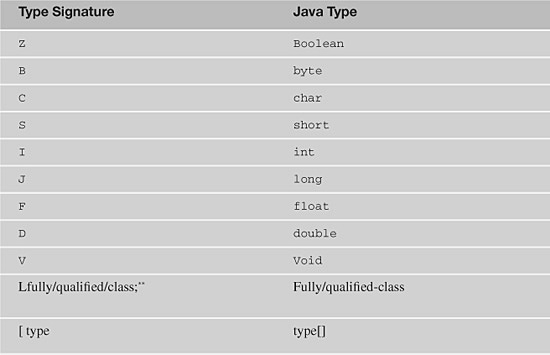
**Manipulating SourceGrid Cell Objects**

In addition to data capture, another important implementation task you might encounter is a manipulation of a GUI object. This might mean any of the standard GUI actions, such as entering data into an object, dragging an object, or clicking an object. In addition to presenting data in a grid format, the sample SourceGrid has GUI objects embedded in its cells (checkboxes, dropdowns, and links), and these cells can be employed to illustrate how to use the Rational Functional Tester API to manipulate GUI objects instead of data capture.

JNI Type Signatures

Method signatures for invoke() are indicated using *Java Native Interface (JNI)* type signatures. (JNI is Java’s mechanism for connecting into “native,” or C, platform code.) JNI method signatures use the Java Virtual Machine’s [JVM] type encoding. The signature formalism is summarized in [Table 10.1](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10tab01).

**Table 10.1** JNI Type Signatures[\*](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10tn01)



[\*](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10tn_01) Available from [java.sun.com/j2se/1.3/docs/guide/jni/spec/types.doc.html#597](http://java.sun.com/j2se/1.3/docs/guide/jni/spec/types.doc.html#597)

\*\* Note that the trailing semicolon is part of the signature of any fully qualified class.

These type signatures are assembled into method signatures using the following formalism: (argType1argType2 ... argTypen)returnType. There is no delimiter between types in the signature, as there is in Java syntax. Inclusion of the returnType in the signature is optional, but inclusion of the trailing semi-colon for object types is not. To clarify the usage, here are some examples[\*\*\*](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10fn01) of JNI method signatures and their Java counterparts:

image

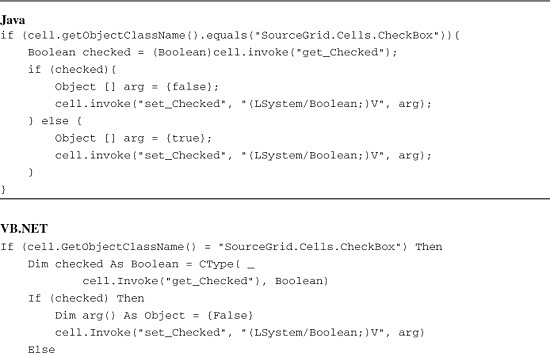
[\*\*\*](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10fn_01) *Java Native Interface: Programmer’s Guide and Specification*, Chapter 12. Sec. 12.3 ([java.sun.com/docs/books/jni](http://java.sun.com/docs/books/jni)).

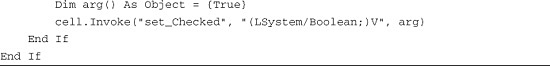
**Checkboxes**

Column 8 of the SourceGrid contains checkboxes instead of text data. There are generally two tasks you need to achieve with checkboxes: data capture to identify the state of the checkbox and direct manipulation of the state of the checkbox. You can capture the state of the checkbox by invoking get\_Value on the cell object (as previously noted); a Boolean is returned that indicates the checkbox state. The following discussion addresses how to alter the state of the checkbox between checked and unchecked.

You can start this task by noting that the class of the cell object for these checkbox-containing cells is different than the class of a regular text-containing cell. A regular text cell is of class SourceGrid.Cells.Cell, but the checkbox cells are of class SourceGrid.Cells.CheckBox (as shown by getObjectClassName()). This suggests that there might be some specific methods for checkboxes in this type of cell, and if you check with getMethodList(), you find that indeed there are. Among its specific methods, there are get\_Checked and set\_Checked. You can use get\_Checked and set\_Checked to manipulate the checkboxes, as shown by the toggle code in [Listing 10.16](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex16).

**Listing 10.16** Manipulating a checkbox





Note that the checkbox object has both get\_Value and set\_Value methods, which return or accept the state of the checkbox as System.Object, not as System.Boolean.

**Links**

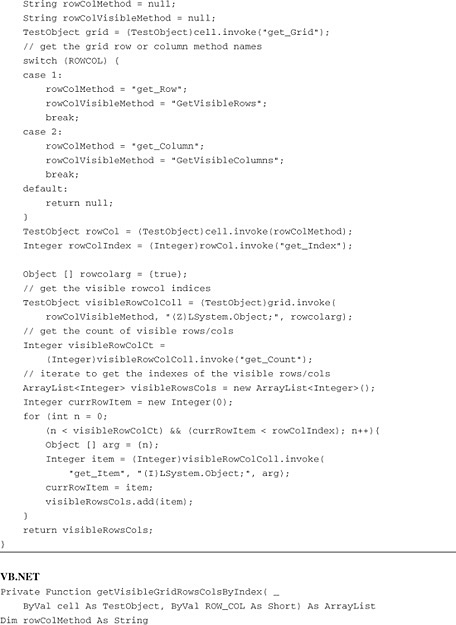
The final column of the sample SourceGrid contains HTML links. For testing purposes, it might be necessary simply to capture the link text and that can be captured, as you saw previously, by invoking the cell’s get\_Value method. However, it might also be required to click a link as part of executing an application transaction. Here, the path you choose depends completely on what methods the cell and grid objects expose. Some cell objects might expose a click() method, although this is unlikely to be part of a cell’s design, and SourceGrid cells do not. More likely, a cell object might expose a method that returns the cell’s coordinates, which can be used to execute a click method. Again, unfortunately, the SourceGrid cell does not. A final strategy would be to iterate through rows and columns and sum their heights and widths to calculate the coordinates of the cell. This supposes, of course, that there are methods to call that enable you to traverse rows and columns and obtain their sizes. However, these types of methods—methods that return row and column collections, row and column objects, and widths and heights—are often part of grid design, and the SourceGrid is no different.

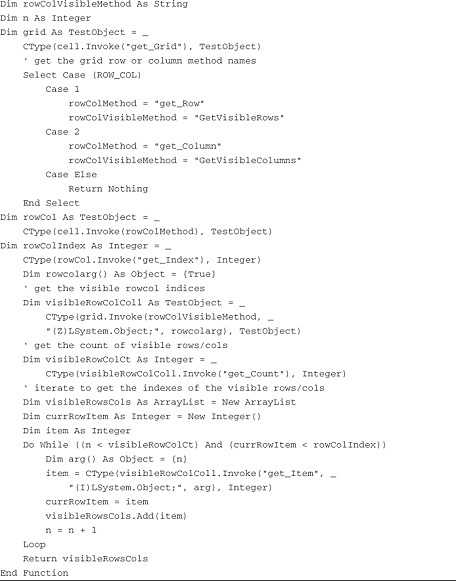
The approach illustrated in the following discussion is to build a small library of methods that can be used to generate the coordinates of the center of a cell, based on the grid’s *visible* rows and columns. (Recall that grids frequently have members that are not currently visible due to the scroll position of the grid’s window.) This library requires a method that generates the visible rows and columns, a method that takes the visible rows and columns as input and calculates the center of a target cell (which must be visible, of course), and a method that generates a click on the cell.

The first of these methods to generate the grid’s visible rows and columns is shown in [Listing 10.17](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex17). The SourceGrid has a convenient set of methods that reports out the visible grid members *by index*. That is, object references to the visible members are not returned—just the index numbers are, the first column being column 0, the second, column 1, and so on, with the rows handled in the same fashion.

**Listing 10.17** Capturing SourceGrid’s visible rows and columns

image

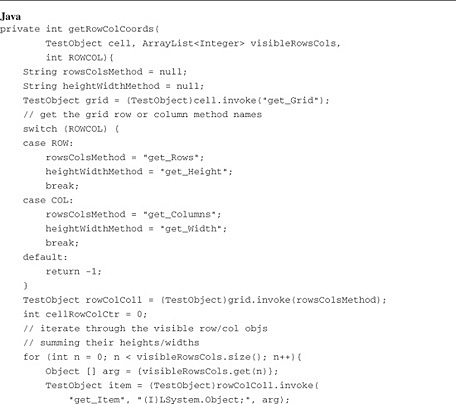


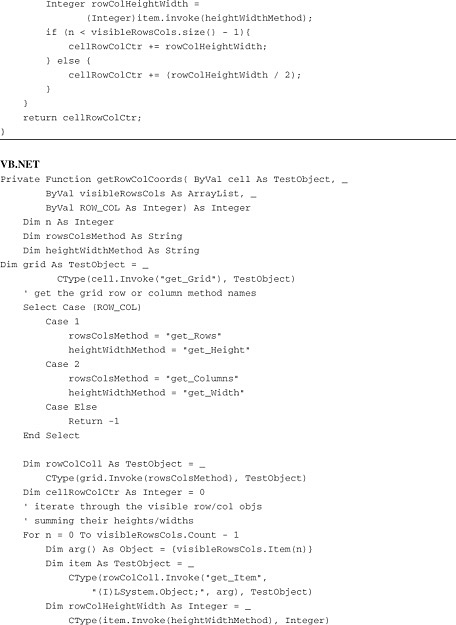


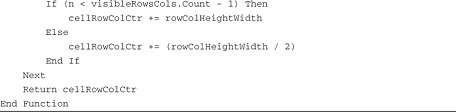
In [Listing 10.17](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex17), the arguments to the method are a reference to the cell TestObject and an int value (ROWCOL) that tells the method whether you want to deal with a row or a column. This ROWCOL argument is used to choose the remote grid methods you invoke.

Now that you have a list of visible rows or columns, a method is needed that can use the visible row or column information, along with the specifics of the cell to determine the coordinates of the center of the cell. [Listing 10.18](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex18) shows how to do this. As with [Listing 10.17](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex17), the code is designed to take a TestObject argument corresponding to the target cell and an int ROWCOL argument that identifies whether the method is to return the horizontal (ROW) or vertical (COL) midpoint of the cell. The method also takes an ArrayList argument that contains the return value from getVisibleGridRowsColsByIndex() ([Listing 10.17](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex17)). The strategy is to access either the rows collection object or the columns collection object (via get\_Rows, get\_Columns) and use the ArrayList of *visible* rows and columns to obtain references to each visible row or column object (get\_Row, get\_Column). The heights or widths of each row or column are used to obtain the vertical or horizontal coordinate of the center of the target cell (with special handling for the dimensions of the target cell itself).

**Listing 10.18** Finding a cell’s vertical or horizontal center

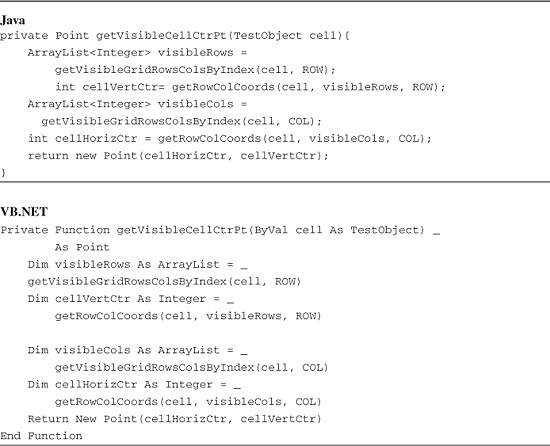






Note that getRowColCoords() in [Listing 10.18](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex18) is designed to return either a horizontal (column) coordinate or a vertical (row) coordinate, so two calls to the method are required to determine the actual center point of the target cell (one for the horizontal coordinate and one for the vertical). This set of tandem calls is shown on [Listing 10.19](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex19).

**Listing 10.19** Calculate the center point of a cell



After obtaining the coordinates of the target cell’s center, there is a final task to complete in order to deliver a click to a cell, which involves noting a subtle difference between the grid and cell references. In [Listing 10.13](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex13), when the grid object is fished out from the form’s controls collection, it is cast as TestObject with no further examination. Similarly, in [Listing 10.14](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex14), when a cell reference is obtained from the grid, the procedure again is to cast it as TestObject with no further investigation. However, if you interrogate each TestObject (grid or cell) using getObjectClassName(), you would find that the grid reference is really a GuiTestObject reference, whereas the cell reference is a TestObject reference. This difference is crucial, because TestObject has no click() method, but GuiTestObject does. Thus, you cannot use the cell reference to click on the cell; you must use the grid reference to do so. This is accomplished by downcasting the grid reference from TestObject to GuiTestObject and using the cell center point (either java.awt.Point or System.Drawing.Point) with the overloaded click() method (in Java):

Point pt = getVisibleCellCtrPt(cell);  
((GuiTestObject)srcGrid).click(pt);

or (in VB.NET):

image

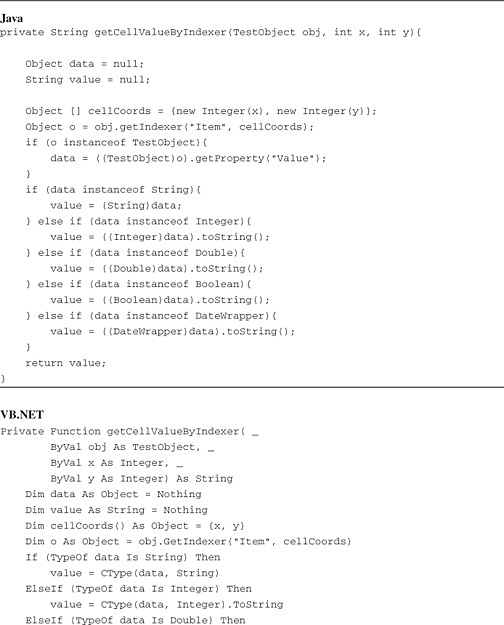
**Listboxes**

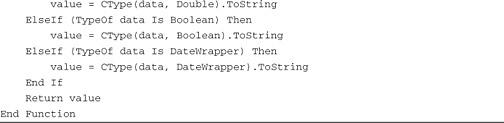
The manipulation of listbox cells in the SourceGrid is a relatively simple matter, because although Rational Functional Tester does not recognize the grid itself natively, it does recognize the listbox. So, the Object Map can be used as normal and recording works as normal. The only issue to be dealt with is how to create the listbox object so that the Rational Functional Tester can manipulate it. In regular use, this is done with a click on the cell, so you can simply use the solution developed in [Listings 10.17](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex17)—[10.19](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex19). Although this code was developed to click on a cell with a link, the code makes no reference to the link itself and can be used to deliver a click to any cell.

**How Many Different Ways Are There to Capture Data from a Control in Rational Functional Tester?**

At this point, you have seen a variety of ways to effect data capture using the Rational Functional Tester API from a target application. In previous chapters, you’ve seen full discussions of how to use getProperty() and the ITestData interface manifold for data capture. In this chapter, the additional option of using the invoke() method to invoke remote data-related methods on the target was added. In addition to these three key approaches, there is an additional one that is not really a different approach but rather is a combination of techniques you have already seen—the use of an indexer to find an object reference followed by getProperty() to capture its data. [Listing 10.20](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch10.html#ch10ex20) shows the approach using the SourceGrid.

**Listing 10.20** Using an indexer with getProperty()





**Summary**

All the handling developed in this chapter gives you the tools to discover target objects dynamically, to work off-Map if you need to, and to capture data from unsupported third-party controls and to manipulate those controls with GUI actions. All these techniques provide extraordinary power for dealing with challenging object architectures that different target applications might offer. However, this chapter’s discussion of third-party controls has not included any mention of how it might be possible to use the full Rational Functional Tester mechanism—including the recording engine—against an unsupported control. This topic is examined in [Chapter 13](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13), “[Building Support for New Objects with the Proxy SDK](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch13.html#ch13),” where you look at the Rational Functional Tester Proxy SDK.

**For Further Reading**

Liang, S. *Java Native Interface: Programmer’s Guide and Specification*. Reading, MA: Addison-Wesley Pub. Co., 1999.

Gordon, R., Gordon, R., McClellan, A. *Essential JNI*. Upper Saddle River, NJ: Prentice Hall, 1998.

[Prev](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html)

[Chapter 9. Advanced Rational Functional Tester Object Map Topics](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch09.html)

[Next](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch11.html)

[Chapter 11. Testing Specialized Applications](https://www.safaribooksonline.com/library/view/software-test-engineering/9780137036455/ch11.html)