Java™ Technology Test Suite Development Guide 1.2

For Java Compatibility Test Suite Developers

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650-960-1300

November 2003

UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et licenciée exlusivement par X / Open Company, Ltd.

Le logo Adobe® est une marque déposée de Adobe Systems, Incorporated.

Please

Contents

Figures

Preface

This document describes methods to develop a test suite for a Technology Compatibility Kit (TCK). A TCK is defined as the suite of tests, tools, and documentation that allows an implementor of a Java technology specification to determine if the implementation is compliant with the specification.

Who Should Use This Book

This book is meant for software engineers who are developing Java compatibility tests suites for a TCK. Its purpose is to assist expert groups working within the worldwide Java Community Process℠ (JCP℠) program to create API tests for a new TCK.

How This Book Is Organized

Chapter 1, "Introduction" provides a brief introduction on how to use this document.

Chapter 2, "Test Development Processes and Infrastructure" describes the processes, infrastructure and steps in developing compatibility tests.

Chapter 3, "Test Case Planning and Design" describes the specialized requirements and techniques for specifying API test cases for the purpose of planning and designing compatibility tests.

Chapter 4, "Analyzing Java API Specifications" describes the requirements and techniques for analyzing API specifications for the purpose of developing compatibility tests.

Chapter 5, "Writing Java API Compatibility Tests" describes the techniques used by Sun for writing Java API compatibility tests.

Chapter 6, "Writing Tests for Execution by a Test Harness" describes how to write tests which will be executed by a test harness.

Chapter 7, "Test Integration and Quality Assurance (QA)" describes the general processes and procedures used at Sun to integrate compatibility tests into a TCK and perform QA testing of the TCK product.

Chapter 8, "TCK Maintenance" describes techniques for improving tests, analyzing test failures from the field, and excluding tests.

Appendix A, "Test Writing Exercise Answers" provides answers to the test writing exercises.

Appendix B, "HTML Test Description Code Listings" lists the actual test description code from the examples and exercises.

Appendix C, "Introduction to Java Technology API Specifications" briefly introduces the structural levels of an API specification.

"Java TCK and CTT Glossary" contains definitions of key terms for all Java CTT products.

Typographic Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`% su` `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this.
	Command-line variable; replace with a real name or value	To delete a file, type `rm` *filename*.

Related Documentation

To become familiar with the formal process of developing a Java specification within the Java Community Process℠ program, see the following URL:

`http://www.jcp.org/`

Note – References to specific Web URLs are subject to change.

Also see the following support and tool documentation which are included in the Java CTT distribution:

- *TCK Project Planning and Development Guide*
- *Java Technology Compatibility Kit User's Guide Template*
- *Spec Tool User's Guide*
- *Signature Test User's Guide*
- *Java API Coverage Tool User's Guide*
- *JavaTest User's Guide and Reference*

Note – The top most Java CTT installation directory is referred to as *CTT_HOME* throughout this document.

The following titles are a good basic resource on the Java programming language:

- *The Java Programming Language*
- *The Java Language Specification Second Edition*
- *The Java Virtual Machine Specification 2nd Edition, Java 2 Platform*

They can be found at this URL:

`http://java.sun.com/docs/books/`

Accessing Sun Documentation Online

The Java Developer Connection℠ web site enables you to access Java™ platform technical documentation on the Web:

`http://developer.java.sun.com/developer/infodocs/`

Sun Welcomes Your Comments

We are interested in improving our documentation and welcome your comments and suggestions. You can email your comments to us at:

docs@java.sun.com

CHAPTER 1

Introduction

This chapter briefly describes how to use this document to learn the concepts and techniques used to develop Java compatibility test suites for a TCK product.

Before using this document, you should be familiar with the general guidelines for TCK development according to the Java Community Process (JCP) program as described in the *TCK Development Guidelines*. See "Related Documentation" in the Preface.

The chapters in this document are presented in an order most beneficial to readers new to TCK test suite development techniques.

This document will also be helpful to those who need an overview of test development techniques for planning or management purposes.

Detailed reference material describing the techniques used at Sun to develop TCK test suites is included for those writing test code. This document also includes test writing examples based on sample code, and hands-on exercises. If you will be developing tests for a finished TCK, you should read this entire document in the order presented and complete the test development exercises.

For a complete summary of the topics covered see "How This Book Is Organized" on page xi.

CHAPTER 2

Test Development Processes and Infrastructure

This chapter describes the processes and infrastructure that could be used to develop compatibility tests for a new API library.

The entire process of developing compatibility tests for a fairly large API is usually a multi-month process that includes the development of a number of related components. This chapter describes the process along with the various components that can be used to help plan a large scale project and make it more manageable.

Compatibility testing differs from product testing which can begin as soon as some part of the program is written. Compatibility testing requires both a working implementation and an accurate, complete specification in order to complete the test development process.

A large-sized test suite might include many thousands of tests. This requires using a test harness to manage the test execution and reporting of results. Compatibility testing at Sun is accomplished using its JavaTest™ test harness and tools. As a result, this document uses the JavaTest harness when depicting examples of test harness usage.

Process Management Components

The following bullets briefly list the components used by many test organizations to manage and implement a large test suite development project. Subsequent sections describe them in more detail. In addition, other chapters will further refine the process of developing a number of these components. Note that some of these components might be considered required and some optional, depending on the complexity of a particular project.

The components used at Sun within the JCP program to manage and implement a large scale compatibility test suite project are as follows:

For planning

- Accurate and complete API specification
- Test plan document
- Test design document
- Test integration plan

For test suite development

- Test case specifications
- Test descriptions (for the JavaTest harness)
- Tests

Accurate and Complete API Specification

Compatibility testing is based on a written specification of the technology. Fundamentally, it verifies that the behavior of an implementation of the specification conforms to behavior that is accurately and completely described in the specification. Developing a compatibility test suite begins with a written specification document that is both accurate and complete.

Note that Sun recommends that a Java technology-based API specification be defined as the Javadoc™ tag comments for a given API, and any supporting documents that are linked into the Javadoc tag comments.

Chapter 4, “Analyzing Java API Specifications” further describe show to analyze a specification for accuracy and completeness.

Test Plan Document

Responsibility for development of the required TCK is assigned to the Specification Lead of the related Expert Group within the JCP program. Once the specification is drafted, a test plan document should be written to describe how the tests within the TCK test suite will be developed. The test plan document should provide a comprehensive description of the TCK as a test suite product, including these items:

- What is being tested by the TCK.
- Who is responsible for developing tests.
- Who is responsible for ongoing maintenance of the tests.
- What resources are required to develop the TCK test suite.
- What are the milestones for test development and product development.
- What are the risks.

Test Design Document

Test design documentation addresses the following types of factors:

- Any items to be included in the API implementation under test in order to execute the tests, such as any special TCK related hardware or software.
- The types of tests that are being developed and whether they will be differentiated based on class, API functionality, or other test classification characteristics.
- The level of test coverage that will be completed by the planned product shipment date. Ideally, the goal is to write both class-based and functionally-based tests for the entire implementation of the Java technology API specification.
- The type of test development that will be proposed for the API, described in detail, including sources for test case development methodology and completeness metrics.
- How the test results will be verified.
- How test cases or areas of functionality to be tested will be grouped into tests for the test suite.
- Any necessary enhancements to the existing test development or test execution tools that are being used; for example, if there is a need for unique or specialized hardware due to an unusual characteristic or limitation of the technology under test.
- Any issues that must be addressed due to a special TCK related hardware or software characteristic, such as support of a limited number of files or limited file name length.
- Any changes in the usual testing process that will be required by the planned test suite.
- Any planned standard test library code that will be needed.

It may not be necessary to provide a separate test design document in every case. If the test design information is not very complicated or lengthy, it can optionally be put in either the test plan or the test specification documentation, described in "Test Case Specifications" on page 6. However, for the purpose of clarity it should not be divided between both forms of documentation.

If you are developing compatibility tests that are similar to existing tests developed for a previous version of a technology, it is acceptable to provide the test design information in the test plan document. However, if you are writing test types that have not been developed for a compatibility test suite before, you should provide a separate test design document.

Test Integration Plan

Test integration is the process of incorporating individual test cases into the larger whole of the test suite. Options include either making each test case an individual test, or grouping related test cases into a test.

The integration plan will be dependent on the software development processes and practices at a particular test development site. For example, when using the JavaTest harness Sun follows these guidelines. All of the tests, test case specifications, and test descriptions are developed in a prescribed format that can be easily integrated into the appropriate compatibility kit workspace. The workspace stores the files used to develop, compile and build the test suite. TCKs developed by Sun are usually distributed as a single ZIP file archive containing the test suite, test harness, and all associated documentation.

The integration process that is used depends on the requirements of the test harness in use. In general, the integration stage is first accomplished through carefully planned procedures, and then verified for accuracy and completeness with some related integration tests. Both the integration procedures and the tests to be used for this purpose should be appropriately documented in some form.

See Chapter 7, “Test Integration and Quality Assurance (QA).”

Test Case Specifications

Test case specifications describe in logical terms what a test does and the expected results. A test case specification refers to one or more assertions that a particular test case is to verify, as derived from the related API specification. For a detailed description see Chapter 4, “Analyzing Java API Specifications.”

If adequate test case specification information can be incorporated into the test design document, then it is not necessary to develop exhaustive test case specifications for each test in separate documentation.

A specific test harness will usually exhibit its own documentation placement options. For example, when using the JavaTest harness with a test suite it is recommended to provide the individual test case specifications in HTML within the test suite test directory tree (because it is portable). If they are straightforward then you can locate them within the same HTML file as the test description, discussed next; if not, use separate HTML files with links to the appropriate test description files. Writing test case specifications is discussed further in Chapter 3, “Test Case Planning and Design.”

Test Descriptions with the JavaTest Harness

In this context, *test descriptions* are defined as the structured HTML or Javadoc tag comments that are used by the JavaTest harness when executing the tests. Another test harness might use a different method to obtain the data necessary to run individual tests. When using the JavaTest harness, all tests, test case specifications, and test descriptions must conform to the format that it uses.

Tests

Tests are the source code and any accompanying information that exercise a particular feature or part of a feature to perform the compatibility testing functions of a test suite. Some interactive tests are a combination of code and manual procedures.

Writing tests is described in Chapter 5, “Writing Java API Compatibility Tests.”

General Time Line and Planning Considerations

This section lists a number of time line and planning considerations.The estimates loosely describe the time lines for the processes used at Sun. We emphasize that they are only for the purpose of example because they vary across projects.

- The time line for test development depends on the complexity of the API feature being tested. As a rough estimate, budget one engineer-hour in TCK test development for each engineer-hour spent on developing the technology implementation. If the implementation is not complicated, also include engineering time spent on design.
- Get the test development team trained in compatibility testing early. The sooner your test development team is trained about compatibility testing, the better they will be able to leverage their own test development work.
- Deliver the test plan for review to the responsible parties as early as possible in the development cycle, and encourage feedback.
- On a large scale project, consider having the test design decided upon and reviewed about 3–4 months before final source code shipment.
- Attempt to ensure that the specification team and the reference implementation development team contact the test suite development team early and often with design issues. Keep test suite developers informed if there are changes in what needs to be tested, or if issues are arising.
- Have a test ready for formal inspection several months before the technology release date. We recommend having the first test inspected as soon as it is ready as there could be much re-work required which could affect the schedule.
- An inspection and review process is usually scheduled as a series of in-depth meetings held over a one to two week period. A thorough source code inspection or review will generally take a significant amount of time. However, a well executed inspection or review will reveal many significant issues, and is well worth the time.
- At around the first inspection, begin integration into the appropriate test suite workspace. Initial time required for this might be about one week.

- At Sun, we start running what is termed *partial builds* of the compatibility kit in order to test selected parts of the build process for errors. This starts close in time to the first inspection and after workspace integration begins. The initial time required for setting this up is about one week, but this and the exact method of running a partial build depends on the build process in use. Partial builds allow continual testing as development progresses, uncovering problems earlier in the cycle.
- On a daily basis during the last month before source code shipment, we also update the test suite and run partial builds of the compatibility kit. This makes sure it is functioning properly.
- During the last month before the finished product ships, have a final review by the test suite development team. This should go fairly quickly if these recommendations have been followed as outlined. This final review will include verifying the correctness of the test suite contents, and its packaging and test coverage.

Within the JCP program a technology compatibility kit is released at the same time as the related reference implementation and specification.

CHAPTER 3

Test Case Planning and Design

This chapter describes the specialized requirements and techniques for specifying API test cases for the purpose of planning and designing compatibility tests.

Behavior-Based Testing of the Specification

Compatibility testing is a *behavior-based* testing activity. Is purpose is to test written *specified* behavior with a minimum knowledge or concern for the underlying structure of the code. Other synonyms for behavior-based testing are:

- Black-box testing
- Functional testing
- Data-driven or input/output-driven testing

In direct contrast to behavior-based testing, *structure-based* testing requires a knowledge of the logic of the code to guide the selection of test data without the specification in mind. Other synonyms for structure-based testing are white-box testing or internals-based testing. Some of the methods used to test code directly are:

- Statement coverage
- Decision coverage
- Condition coverage

This document deals only with the behavior-based testing strategies collectively known as compatibility testing.

Compatibility Testing vs. Product Testing

Compatibility testing differs from traditional product testing in a number of ways.

- Unlike product testing, compatibility testing is not primarily concerned with robustness, performance, or ease of use.
- The primary purpose of compatibility testing is to determine whether an implementation of a technology is compliant with the specification of that technology.
- The primary goal of compatibility testing is to provide the assurance that an application will run in a consistent manner across all implementations of a technology that were certified as compatible.
- Compatibility test development for a given feature relies on a complete specification and reference implementation for that feature.
- Compatibility testing is a means of ensuring correctness, completeness, and consistency across all implementations of a technology specification that are developed.

The fundamental differences between TCK tests and product tests are further summarized in TABLE 1.

TABLE 1 TCK testing versus product testing

TCK Testing	Product Testing
Compatibility testing is behavior-based. It tests written specified behavior with a minimum knowledge or concern for the underlying structure of the code. Termed black-box or functional testing, or data-driven or input/output-driven testing.	Product testing is structure-based. It relies on a knowledge of the logic of the underlying implementation specific code to guide the selection of test data. It does not consider the specification. Termed white-box or internals-based testing.

TABLE 1 TCK testing versus product testing

TCK Testing	Product Testing
Typical TCK tests do not contain any stress tests or performance measurement code. Performance and other related limitations can vary with the particular implementation of the technology. This type of testing is usually outside the scope of the Java technology specification. An exception is when the related Java technology specification clearly defines certain limitations or requirements on resources. In this case the TCK should test these assertions accordingly.	Product performance tests should attempt to make full use of all available resources. They attempt to reach certain predefined limitations on the particular implementation under test. For example, stress tests might include opening a large number of windows, files or sockets, or launching a large number of threads or processes. They might also attempt to allocate all or most of the available memory.
TCK tests must be written for a wide spectrum of implementations. They are used on potentially different operating systems with varying requirements for resources. Predefined configuration parameters are typically beyond the scope of the Java technology specification; no assumptions can be made about them in TCK tests.	Product tests are typically written for a specific software product. They can make assumptions about how the product is configured and running based on consistent values—for example, in configuration files and command-line parameters. The product itself typically has predefined requirements, such as a specific underlying OS or CPU, or specific memory or disk space allocations. Product tests can rely on this product-specific information.
TCK tests are validated by running them against a reference implementation (RI). Note that TCK tests should not rely on any RI-specific resources or configuration information. This includes items such as file or class names and locations, or other implementation-specific features or details.	Product tests are validated using a particular version of the product and should fully test product-specific features and details.

Test Development Strategy

As previously described, API compatibility testing is based on testing the behavior that is specified by the assertions made in the API specification. Compatibility test development involves these basic steps:

1. Identify all the assertions contained in the specification.
2. Develop the appropriate test cases for each assertion.

Step 1: Identify Assertions in the Specification

An assertion is a statement that specifies some necessary aspect of the API. It is a statement that a developer must adhere to when implementing the specified Java technology. It is also a statement that application developers can rely upon. Examples of testable assertions in a specification are statements such as *Returns the name of the stock* or *Constructs a GetQuoteException with the specified detail message and nested exception*.

Assertions are critical to both the implementation of a Java technology specification and its subsequent compatibility testing. At this point in test development, it is critical that assertions have been clearly and concisely stated in the finished specification. They must be stated so that implementors, application developers, and compatibility test developers all derive the same meaning from them. Details on how to identify well defined assertions in the specification are covered in Chapter 4, “Analyzing Java API Specifications.”

Note – The Java programming language is strongly typed. It is not necessary to test data type assertions for method arguments and return values. You can assume that type checking of input and output conditions is performed by a correctly functioning Java compiler and / or Java virtual machine.

Analyzing specification assertions is described in more detail in Chapter 4, “Analyzing Java API Specifications."

Step 2: Develop Test Cases for the Assertions

A *test case* is the source code and accompanying information designed to exercise one aspect of a specified assertion. Accompanying information may include test documentation, auxiliary data files and other resources used by the source code.

In order to be complete, a test suite must include a test case to verify each and every testable assertion that is made by the API specification. Test developers should review the actual specification document and generate at least one test case for each testable assertion that appears in the API specification.

How to recognize testable assertions is further discussed in Chapter 4, “Analyzing Java API Specifications.”

Test cases are usually documented during development within the test case specification sections of the finished TCK. In some instances, specification assertions and their resulting test cases are combined in a separate document.

Developing test cases for assertions is described in Chapter 5, “Writing Java API Compatibility Tests."

The Java CTT distribution includes the following tools and documentation to assist in identifying and tracking specification assertions, and measure the completeness of TCK test coverage. See the user’s guide included in the Java CTT distribution for details:

- **Java API Coverage Tool.** Provides a quick way to estimate TCK test coverage without doing any additional assertion markup when the TCK tests and the specification are under development. It is also useful to gauge completeness of a finished TCK.
- **Spec Trac Tool.** Helps to identify, classify, and track assertions and their related tests.

CHAPTER 4

Analyzing Java API Specifications

Compatibility testing demonstrates either compliance or lack of compliance to a written specification. An accurate and complete specification is the prominent requirement, and without it meaningful compatibility test cases cannot be developed. This chapter describes the requirements for analyzing API specifications for compatibility testing of a Java technology (see footnote[1]).

There has been much discussion about exactly what a specification for a Java technology API library should consist of. Sun recommends that the Javadoc tag comments comprise the specification for each Java technology API library, along with any additional supporting documentation that is referenced as necessary.

For information on the Sun conventions for writing Javadoc tag comments, see *How to Write Doc Comments for the Javadoc™ Tool* on the Web at this URL:

`http://java.sun.com/products/jdk/javadoc/writingdoccomments.html`

For a brief introduction to the basic components of an API specification see Appendix C, "Introduction to Java Technology API Specifications."

Note that an API under test and the implementation that it is part of must provide enough resources to enable testing. Resources include things like the following:

- Input and output that both the JavaTest harness and a test can use to communicate with each other
- Enough methods on the API under test to allow tests to interact with the API

For example, consider the situation if a String object had no methods, but only constructors. In this case, we would not be able to test whether the String object was created correctly because we would have no way of getting any data from it.

1. The fundamental principles covered in this section are based on the Object Class Specification by Edward V. Berard, Essays on Object-Oriented Software Engineering, 1993 Simon & Schuster, Englewood Cliffs, NJ; pp. 131-162.

Specification Assertions Must Be Clearly Stated

API compatibility testing is based on testing the behavior that is specified by the assertions made in the API specification. Compatibility test development involves identifying all the testable assertions made in the specification, and formulating appropriate test cases for each of them.

An assertion is a statement that specifies some necessary aspect of the API. A developer must adhere to an assertion when implementing the Java technology that is specified. An assertion is also a specification statement that application developers can rely upon.

Because of this, assertions are critical to both the implementation of a Java technology specification and its subsequent compatibility testing. They should be clearly and concisely stated so that implementors, application developers, and compatibility test developers will all derive the same meaning from them.

Examples of testable assertions in a specification are statements such as *Returns the name of the stock* or *Constructs a GetQuoteException.*

Identifying Specification Assertions

Prior to the test design stage, the test developer should scan through the specification and split the entire text into logical statements. Each logical statement should then be examined by type to indicate if it is a testable assertion.

The Java CTT distribution includes the Spec Trac Tool which helps to identify, classify, and track assertions and their related tests. Using Spec Trac you can mark up assertions and save the specification results with the following attributes:

- Testable
- Required
- Implementation-Specific
- Ambiguous
- Not an assertion

The following sections describe these assertion attributes here as they relate to test development. Also see the *Spec Trac Tool User's Guide* included in the Java CTT distribution.

Testable

Statements are considered testable assertions if they are intended to describe any behavior of the API which can be tested by the TCK.

- **Example 1:** `java.lang.Integer.toString(int i, int radix)` method description

  "If the radix is smaller than `Character.MIN_RADIX` or larger than `Character.MAX_RADIX`, then the radix `10` is used instead."

  Test development for this sample assertion is described in detail in "Example 1: TCK Tests for `Integer.toString(int, int)`" on page 33.

- **Example 2:** `java.lang.Class.getModifiers()` method description

  "Returns the Java language modifiers for this class or interface, encoded in an integer."

  Test development for this sample assertion is described in detail in "Example 2: TCK Tests for `Class.getModifiers()`" on page 37.

Testable Assertions in Examples or Sample Code

Statements forming examples or sample code pieces that are provided in the specification are typically testable and should be verified by the TCK. In this sense, examples or sample code are generally considered testable assertions.

- **Example 1:** `java.lang.Integer.parseInt(String s, int radix)` method description

  "parseInt("1100110", 2) returns 102"

  A TCK test can be developed to verify that the `parseInt("1100110", 2)` method call returns an Integer whose `int` decimal value is 102.

- **Example 2:** `java.lang.Class.forName(String className)` method description

  "For example, the following code fragment returns the runtime `Class` descriptor for the class named `java.lang.Thread`:

  ```
  Class t = Class.forName("java.lang.Thread")
  ```

  A TCK test can be developed to verify that the `Class.forName("java.lang.Thread")` method call successfully completes and the value of the `t` variable is a Class object representing the `java.lang.Thread` class. Possible ways to verify this are to check that `t.getName()` returns `"java.lang.Thread"` and `t.newInstance()` creates a new instance of a `java.lang.Thread` class.

- **Example 3:** `Quote.convert(int)` method description

  "Convert an `int` to a String with the decimal placed back in (divide by 10000).

Example: -100 -> ‘-0.01’ ”

A TCK test can be developed to verify that the convert(-100) method call returns a string with value "-0.01". See sampleTCK tests for `Quote.convert(int)` at this location in the Java CTT (test `Quote2008`):

`CTT_HOME/examples/sampleTCK/tck/tests/api/com_sun/tdk/sampleapi/Quote/index.html#Public`

- **Example 4**: `Quote.makeInt(string)` method description

“Takes a String representation of a floating point number and makes an `int` out of it.
...
Example: 345.67 -> 3456700 (/10000 = 345.67)”

A TCK test can be developed to verify that the `makeInt("345.67")` method call returns an Integer whose `int` decimal value is 3456700. See sampleTCK tests for `Quote.makeInt(string)` at this location in the Java CTT (test `Quote2009`):

`CTT_HOME/examples/sampleTCK/tck/tests/api/com_sun/tdk/sampleapi/Quote/index.html#Public`

Required

A Required assertion indicates simply that the functionality must be implemented. This is independent of the other attributes as follows: a Required functionality may or may not be Testable, and it may or may not describe Implementation-Specific behavior, and it may or may not be Ambiguous.

Implementation-Specific

Implementation-Specific assertions occur when the precise details of the behavior are deliberately unspecified; how they are implemented is left to the discretion of the implementor. This attribute is independent of the other attributes in that it may or may not be Testable, or Required, or Ambiguous. If Required is FALSE and Implementation-Specific is TRUE, then you can decide whether to implement it and how.

- **Example 1**: `java.awt.Component.paramString()` method description

“Returns a string representing the state of this component. This method is intended to be used only for debugging purposes, and the content and format of the returned string may vary between implementations.”

This behavior is specified as implementation-specific and therefore this assertion cannot be tested.

Statements intended to describe the behavior of the API but which cannot be tested by the TCK due to the special nature of the behavior or functionality are considered nontestable assertions.

- **Example 1:** `java.lang.Runtime.exit(int status)` method description

  "Terminates the currently running Java Virtual Machine. This method never returns normally."

  It may be impossible to verify whether this method call terminated the Java Virtual Machine or returned normally because of test harness or security characteristics, or other restrictions. Thus this assertion is considered nontestable.

- **Example 2:** `java.lang.Runtime.traceInstructions()` method description

  "If the `boolean` argument is `true`, this method suggests that the Java Virtual Machine emit debugging information for each instruction in the Java Virtual Machine as it is executed."

  This documented behavior is not required, and therefore this assertion cannot be tested.

Ambiguous

An ambiguous assertion is one that is not testable. It indicates a bug in the specification that should be fixed by the specification developer. If Ambiguous is TRUE then Testable must be FALSE. An Ambiguous assertion can also describe Required or Implementation-Specific behavior with all marked TRUE.

Not an Assertion

Statements classified as "Not an Assertion" form general descriptions of the API which do not describe behavior, but are aimed at providing a context for the rest of the text. These general descriptions are not intended to be tested. They include items such as a general description of a package, class, method, or field, and so forth. Be careful not to consider these statements to be assertions, as they are easy to misinterpret as such.

- **Example 1:** `java.lang.Integer` class description

  "The Integer class wraps a value of the primitive type `int` in an object."

  Although it might be misinterpreted as an assertion, this statement is not intended to assert that the Integer class instance is an object. Actually this is self-evident because all class instances are objects, and this is enforced by the Java Compiler and the Java Virtual Machine. It is also not intended to assert that it provides a certain kind of access to its only field of type integer. This

behavior is covered in assertions in the specification which describe the particular requirements for the access methods and their arguments and return types.

- **Example 2:** `java.lang.Class` class description

  "There is no public constructor for the class `Class`."

  Absence of the public constructor is evident from the rest of specification. There is no need to develop tests for this type of statement.

- **Example 3:** `StockSymbol.getVisualComponent()` method description

  "Subclasses may override this method to include additional data such as charts."

  This is a note describing a possible use of the API by application developers. It does not describe any behavior of the API itself.

Insufficient Specification Coverage

Sometimes a certain aspect of behavior or functionality is not sufficiently covered by the specification to the extent that the same specification can be implemented in several ways. This is considered a specification flaw or bug which needs to be reported to the specification developers. No test can be developed for functionality that is insufficiently specified.

The following are examples of incomplete specifications taken from JDK 1.1 which were fixed in JDK 1.3.

- Example 1: `java.lang.Integer.toHexString(int i)`

  "Creates a string representation of the integer argument as an unsigned integer in base 16.

  The unsigned integer value is the argument plus 2^{32} if the argument is negative; otherwise, it is equal to the argument. This value is converted to a string of ASCII digits in hexadecimal (base 16) with no extra leading `0`s."

  It is not clear here whether lowercase or uppercase letters should be used for hexadecimal representation. In this case, the following additional assertion is necessary to develop tests:

  "The following characters are used as hexadecimal digits: 0123456789abcdef"

- Example 2: `java.lang.Class.getName()`

  "Returns the fully-qualified name of the type (class, interface, array, or primitive) represented by this Class object, as a String."

  It is not clear what should be returned for the array type since arrays do not

have fully-qualified names.

The following assertion is needed to cover the case of arrays:

“If this `Class` object represents a class of arrays, then the internal form of the name consists of the name of the element type in Java signature format, preceded by one or more "[" characters representing the depth of array nesting.”

Implied Assertions

Some assertions can be implied or indirectly stated in the specification, and these should be identified as well for testing. Note that an implied assertion might also introduce a specification flaw that may not be obvious.

Example: `java.lang.Integer.byteValue()` method description

“Returns the value of this Integer as a byte.”

This assertion implies that standard narrowing primitive conversion as defined in *The Java Language Specification 1.0*, Section 5.1.3, will be applied when casting an `int` value into a byte. Even though the specification for this method does not explicitly state which kind of conversion will be applied, it is pretty safe for the test developer to assume narrowing primitive conversion since no other ways to convert `int` to a byte are defined by the Java platform.

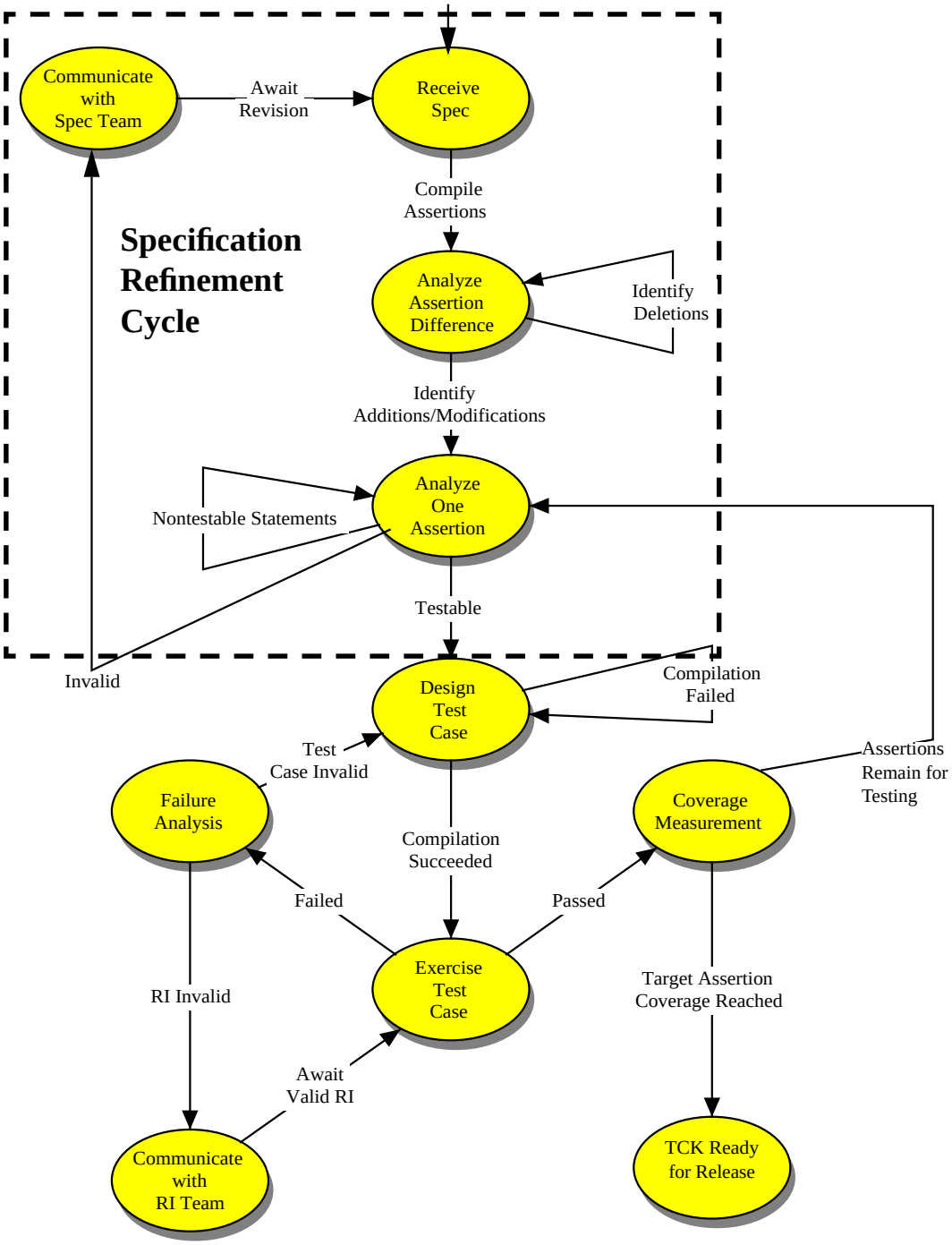
Refining the Specification

It is not always possible to get a complete and accurate specification on the first iteration. A major priority when developing compatibility tests is to uncover any omissions or contradictions (bugs) in the specification and report them to the specification team while it is still in the review cycle.

Omissions or contradictions in the specification are sometimes difficult to uncover without very close and objective inspection. In addition to the formal inspection of specification documents, there should always be a system in place to report and investigate any bugs in the specification before it is finalized. This is done both before and during the test development process.

The compatibility test development team should work closely with the specification team in refining the early drafts of the specification in order to accomplish these objectives. The process of refining the specification is accomplished in an iterative fashion with the specification team working closely with the test development team. This iterative process is illustrated in FIGURE 1 on page 23 within the section of the test development state flow diagram enclosed by broken lines.

FIGURE 1 Refining the specification within the test development process



CHAPTER 5

Writing Java API Compatibility Tests

This chapter describes the techniques used at Sun for developing Java API compatibility tests for a TCK product developed under the JCP program. We assume that you are already familiar with the concepts and techniques presented in previous chapters.

To build a firm background for approaching test development, we suggest that you familiarize yourself with the following industry standard documents:

- IEEE Std. 829-1983, IEEE Standard for Software Test Documentation
- IEEE Std. 1044-1993, IEEE Standard Classification for Software Anomalies
- IEEE Std. 1044.1-1995, IEEE Guide to Classification for Software Anomalies

You should also be familiar with the test development processes and tools used within your own organization. This includes the following TCK development issues:

- Test development guidelines, processes, and procedures
- Test development documentation requirements
- Source code style conventions and guidelines
- Test interfaces for the TCK under development
- Test execution arguments for the TCK under development
- Test description formats for the test execution engine being used
- Test integration processes used in the development group

Test Development Process

Once the preparation work and documentation have been completed as previously described, it is time to write the test code. At this point you should have identified all the testable assertions defined in the specification.

This section uses straightforward steps for the purpose of explanation. In practice, test development is an iterative process of test case development, test writing, and running the tests.

During the process of test development, anomalies in the specification may come to light. These are anomalies that were overlooked during the earlier stage of creating the related test specifications. In this case, the problems are reported to the specification team and related test case development can be postponed until the specification is corrected. See Chapter 4, "Analyzing Java API Specifications."

Writing Compatibility Test Code

This section discusses some of the fundamental requirements that compatibility test development puts on the test code.

Observing Source Code Conventions

Test source code becomes part of the TCK product under the JCP program. End users review the source code whenever necessary in the course of using the TCK to test an implementation. It is important to follow accepted source code style conventions, as well as document the code so it is understandable by reasonably knowledgeable users

The code should not include any rude, indecent, or vulgar method names, variable names, or other text.

These conventions are obvious to experienced developers. However, they are worth mentioning in order to emphasize that a TCK is delivered as a source code product along with its binaries.

Sun provides information regarding source code conventions at this URL:

`http://java.sun.com/docs/codeconv/`

General Source Code Requirements

There are some general requirements when writing test source code:

- All tests must have at least one passing mode and one failing mode (in API tests these are part of the source code of the test).
- All tests must implement the approved test interface for the test suite under development.
- All tests must restore the state of the system under test when the test is completed.

- In general, there should be no hard coding of paths, time-out values, or other hardware specific features.
- No test should depend on a previous test result or action.

The Java programming language is strongly typed. So API tests can assume that data type checking of input and output conditions is performed by a correctly functioning compiler, and they need not test for this. Argument and return type checking is done by the VM and language tests.

Building Robust Portable Tests

Tests must be runnable in all valid implementations of the technology associated with the TCK release. The goal in developing robust portable tests is to eliminate any implementation-specific assumptions or dependencies from the tests.

It is important to avoid making any assumptions in test code about the system that a test is running on, either directly or implied. This can cause problems when running the tests across varying Java technology implementations.

For example, in a Java 2 Platform, Standard Edition (J2SE™) implementation, all tests must be executable in both the application and applet environments. To enable this for J2SE TCK tests, the test interfaces developed by Sun usually include either of the following pairs of methods.

Either:

```
public static main(String[])
public Status run(String[], PrintWriter, PrintWriter)
```

Or:

```
public static main(String[])
public int run(String[], PrintStream)
```

Note – The TCK architect on a development project specifies the interface for the TCK tests.

Tests should not rely on any non-portable software related characteristic, such as a locale dependency, or a specific time zone or encoding scheme.

The following components or subsystems require particular attention when eliminating implementation dependant assumptions in tests:

- Thread scheduling (thread.suspend and thread.resume)
- Network
- File system
- Graphic user interface display or AWT
- Character encoding standard

- Memory
- Multi-process operating system
- Specific OS platform
- Array ordering

If a particular test must exercise one of these subsystems, it is possible to use that subsystem in the test with either of these two methods:

- Set up the test to query the subsystem directly.
- Write the test to interactively query the user about the subsystem before starting the test.

You can also use a test suite configuration wizard to query the user regarding the implementation environment before running the test suite. A wizard is a dedicated utility which interactively queries the user and constructs the proper configuration settings for the test setup. As a example, consider testing in a security constrained environment. Before running the security tests, a configuration wizard might query the user about the state of any applicable security constraints and then run the tests accordingly.

Test Case Development Techniques

This section describes the two techniques used at Sun to develop compatibility test cases for a Java technology API. They are:

- Equivalence Class Partitioning
- Boundary Value Analysis

It is not necessary to apply these techniques in any sequence.

There are other test case development techniques which are quite acceptable, but they are not currently used at Sun.

The two techniques sometimes produce overlapping test cases. Rather than duplicating test cases, a convenient means of referring to the same test case should be provided within the test case documentation.

Because each Java API is defined in terms of classes, test development is be centered around individual classes. This is essentially behavior-based, unit testing. For those classes that cannot stand on their own, testing a small cluster of functionally grouped classes is adequate.

Equivalence Class Partitioning

The technique of *equivalence class partitioning*[1] entails dividing a large number of potential test cases into smaller subsets with each subset representing an equivalent category of test cases. Each subset is based on some equivalent condition that exists in the larger group of test cases. The reasonable assumption is that the single test case will exhibit the same behavior as the entire category of test cases would if they were all tested under the same conditions. In this manner, a single test case can represent a larger number of test cases, lessening the testing load while providing the same degree of completeness.

There should be one test case for each equivalence class developed from this process.

Note – When using the term equivalence class partitioning, *class* does not refer explicitly to a Java programming language class, but rather to a category of similar test cases.

Equivalence class partitioning strives to reduce the number of test cases needed by dividing these five classes (categories) of data into representative classes of data.

- Input values
- Output values
- Pre-condition data values
- Post-condition data values
- Object states

Once the data is partitioned into equivalence classes, only one or two representative elements of the classes are explicitly tested. The theory is that if no error is found by a test of one element of a set, it is unlikely that an error would be found by a test of another element of the set.

There are also two basic types of equivalence classes which result in either a *positive* or *negative* testing approach:

- **Valid equivalence classes**: test case subsets which contain proper, expected, and otherwise *normal* situations with respect to the object, such as:
  - Typical states
  - Expected values for method input parameters
  - Normal external conditions
- **Invalid equivalence classes**: test case subsets which contain improper, unexpected, and otherwise abnormal situations with respect to the object; these test cases result in *negative tests, which pass only if execution fails in a specified manner*, such as:

  States which are not allowed
  Illegal values for method input parameters
  Abnormal external conditions

1. See Myers, Glenford J.; Art of Software Testing, The; John Wiley & Sons 1979; pp. 44-50.

There are five object orientations in which equivalence classes are considered:

- States which an object may assume or be forced into
- Characteristics of method input conditions for the object
- Characteristics of method output conditions for the object
- Characteristics of any external conditions which exist immediately prior to the occurrence of an event in the life of an object
- Characteristics of any external conditions which exist immediately after the occurrence of an event in the life of an object

Equivalence Class Partitioning Example: `Color (int, int, int)`

Consider the specification for the Java class `Color (int, int, int)`. This class creates a color with the specified red, green and blue values in the range of 0–255.

TABLE 2 identifies three equivalence classes for each of the `int` parameters with a *range* of 0–255, as follows:

- One valid equivalence class (a set of integers 0 to 255)
- Two invalid equivalence classes (those less than 0, and those greater than 255)

TABLE 2 Equivalence classes for each parameter of Color (int, int, int)

Class Type	Description	Equivalence Class Range
Invalid	Minimum int size to -1	-2^{31} to -1
Valid	Specified	0 to 255
Invalid	256 to maximum int size	256 to $2^{31}-1$

Each of the three equivalence class types in TABLE 2 would produce one representative test per class type. This comes to three tests (test cases) for each of the three `int` parameters, or nine tests total.

Testing just one representative element of a set does not provide as much testing completeness as testing around the edges of the set. This degree of completeness is provided by Boundary Value Analysis as a basis for creating further test cases, which is discussed next.

Boundary Value Analysis

Testing just one random element from a set of parameters derived from equivalence classes does not provide sufficient test case coverage. After developing the equivalence class partitioning for the objects to be tested, test developers should also perform *boundary value analysis* by examining the boundaries[1] of the objects within the equivalence classes.

When using boundary value analysis, additional test cases are developed based on the boundaries defined by the equivalence classes. That is to say, testing is done for parameters just above, and just below the perimeters of set values, as well as inside the parameter value bounds.

As with equivalence class partitioning, there are also five orientations in which boundary-values may be considered for objects:

- Boundaries on the states which an object may assume or be forced into
- Boundaries on the characteristics of any input conditions for methods
- Boundaries on the characteristics of any output conditions for methods
- Boundaries on the characteristics of any pre-existing external conditions
- Boundaries on the characteristics of any external conditions which exist immediately after the occurrence of an event in the life of an object

---

Note – This type of testing also helps to uncover many common errors such as the erroneous use of a relation operator. For example, when a *less than* operator is used but a *less than or equal to* operator is meant.

Boundary Value Analysis Example: `Color (int, int, int)`

Continuing with testing the range of 0-255, we would expand the results of equivalence class partitioning shown previously in TABLE 2 and introduce the test values of -1, 0, 255, 256 as shown in the right column of TABLE 3.

TABLE 3 Test values using boundary value analysis

Class Type	Description	Equivalence Class Range	Test Values
Invalid	Minimum int size to -1	-2^{31} to -1	-1
Valid	Specified	0 to 255	0, 255
Invalid	256 to maximum int size	256 to 2^{31}-1	256

1. ibid. pp. 50-55.

However, these values alone will still not exercise thorough combinations of input conditions. It is also important to test invalid parameters individually. For example, it is insufficient to test the `-1` test value with `Color (-1, -1, -1)` to cover all three invalid parameters at once. It is more complete and accurate to test the following three parameter combinations individually:

- `Color (-1, 0, 0)`
- `Color (0, -1, 0)`
- `Color (0, 0, -1)`

As a result of identifying the test cases in TABLE 3, we have four test case values (tests) for each parameter tested individually (no combinatorials).

We stated that testing `Color (-1, -1, -1)` was not alone a sufficient test of invalid parameters. It is nonetheless a useful test to add to the current list along with `Color (256, 256, 256)`. Testing often requires using artful intuition as well as scientific principle.

Not counting the duplicates for `Color (0, 0, 0)`, there is now a total of twelve resulting test cases for `Color (int, int, int)` as shown in TABLE 4:

TABLE 4 Parameters for the `color` Method

Color Parameters			
Color (-1, 0,0)	Color (0, -1, 0)	Color (0, 0, -1)	Color (-1, -1, -1)
Color (0,0,0)	Color (0, 0, 0)	Color (0, 0, 0)	
Color (255, 0, 0)	duplicate	duplicate	
Color (256, 0, 0)	Color (0, 255, 0)	Color (0, 0, 255)	Color (256, 256, 256)
	Color (0, 256, 0)	Color (0, 0, 256)	

Writing the Test Code

This section contains actual examples of TCK tests for several methods from the J2SE API. It describes step-by-step procedures to write them. It also discusses common mistakes that a test developer might make when writing TCK tests.

Note – The examples from this section are simplified for the purpose of clarity. All uses of JavaTest API library classes are removed to make the code independent of any particular test harness. Incorporating the test code into a test harness is described in Chapter 6, “Writing Tests for Execution by a Test Harness.”

Example 1: TCK Tests for `Integer.toString(int, int)`

The following is the API specification being tested:

```
public static String toString(int i, int radix)
```

Creates a string representation of the first argument in the radix specified by the second argument.

If the radix is smaller than `Character.MIN_RADIX` or larger than `Character.MAX_RADIX`, then the radix 10 is used instead.

If the first argument is negative, the first element of the result is the ASCII minus character '-' ('\u002d').

If the first argument is not negative, no sign character appears in the result.

Parameters:

`i` - an integer.

`radix` - the radix.

Returns:

a string representation of the argument in the specified radix.

See Also:

`Character.MAX_RADIX`, `Character.MIN_RADIX`

Based on this specification, you can identify these assertions:

Assertion	Meaning
A1	Creates a string representation of the first argument in the radix specified by the second argument.
A2	If the radix is smaller than Character.MIN_RADIX or larger than Character.MAX_RADIX, then the radix 10 is used instead.
A3	If the first argument is negative, the first element of the result is the ASCII minus character '-' ('\u002d').
A4	If the first argument is not negative, no sign character appears in the result.

Now we use equivalence class partitioning and boundary value analysis techniques to write the actual tests.

The method under test is static, so there is no object state that may affect the return result. Because of this, only input data is used to identify the equivalence classes. For the parameter `i`, this is:

EC1-1: i < 0

EC1-2: i >=0

(See Assertions A3, A4).

For the parameter `radix`, this is:

EC2-1: radix < Character.MIN_RADIX

EC2-2: Character.MIN_RADIX <= radix <= Character.MAX_RADIX

EC2-3: radix > Character.MAX_RADIX

(See Assertion A2).

The appropriate boundary values for `i` are:

Integer.MIN_VALUE, -1, 0, 1, Integer.MAX_VALUE.

The appropriate boundary values for `radix` are:

Integer.MIN_VALUE, Character.MIN_RADIX - 1, Character.MIN_RADIX,

Character.MAX_RADIX, Character.MAX_RADIX + 1, Integer.MAX_VALUE.

Now we add a non-boundary member of each equivalence class to complete a list of input data to perform the testing:

Class	Member
i:	Integer.MIN_VALUE, -1255, -1, 0, 1, 34, Integer.MAX_VALUE.
radix	Integer.MIN_VALUE, -7, Character.MIN_RADIX - 1, Character.MIN_RADIX, 20, Character.MAX_RADIX, Character.MAX_RADIX + 1, 179, Integer.MAX_VALUE.

Since the number of elements is rather small, it is possible to use full iteration on both parameters. Otherwise we would have to select variants manually to prevent combinatoric explosion.

Next, it is possible to calculate the resulting value for each pair `i/radix` by hand and hard code it into the test source. However, the test developer in this case has chosen to write a method based on the above four assertions and use it to dynamically generate the expected result.

Note – An important caveat with this approach is that the actual code for this method must be written independently, in particular, without looking at the reference implementation sources.

This method might look like this:

CODE EXAMPLE 1 Method That Calculates the Resulting Value for each Pair `i/radix`

```
private static String myToString(int i, int radix) {
char[] digits = {
'0' , '1' , '2' , '3' , '4' , '5' ,
'6' , '7' , '8' , '9' , 'a' , 'b' ,
'c' , 'd' , 'e' , 'f' , 'g' , 'h' ,
'i' , 'j' , 'k' , 'l' , 'm' , 'n' ,
'o' , 'p' , 'q' , 'r' , 's' , 't' ,
'u' , 'v' , 'w' , 'x' , 'y' , 'z'};

// Assertion A2
if (radix < Character.MIN_RADIX || radix > Character.MAX_RADIX) {
  radix = 10;
}

char buf[] = new char[65];
boolean negative = (i < 0);
int charPos = 64;

if (!negative) {
  i = -i;
}

// Main loop for assertion A1
while (i <= -radix) {
  buf[charPos--] = digits[(int)(-(i % radix))];
  i = i / radix;
}
buf[charPos] = digits[(int)(-i)];

// Assertions A3, A4
if (negative) {
  buf[--charPos] = '-';
}

return new String(buf, charPos, (65 - charPos));
}
```

```
// We also assume there is a logging method to log messages
// in order to make it easier to debug test failures.

public void log(String s) {
.....
}
```

Now we can write the test method encompassing all test cases for `Integer.toString(int, int)`, as follows:

CODE EXAMPLE 2 Test Method `testToString()`

```
public boolean testToString() {

int[] iValues = { Integer.MIN_VALUE, -1255, -1, 0, 1, 34,
   Integer.MAX_VALUE};

int[] radixValues = { Integer.MIN_VALUE, -7,
    Character.MIN_RADIX - 1, Character.MIN_RADIX,
    20, Character.MAX_RADIX, Character.MAX_RADIX + 1,
    179, Integer.MAX_VALUE};

boolean pass = true;
String expected, result;

  for (int i = 0; i < iValues.length; ++i) {
    for (int j = 0; j < radixValues.length; ++j)     {
       expected = myToString(iValues[i], radixValues[j]);
       result = Integer.toString(iValues[i], radixValues[j]);
         if (! result.equals(expected)) {
            pass = false;
            log("Failed for i=" + iValues[i] + ", radix=" +
            radixValues[j] + "; expected: " + expected +
            "returned: " + result);
          }
    }
  }

return pass;
}
```

Example 2: TCK Tests for `Class.getModifiers()`

This is the API specification being tested.

> Returns the Java language modifiers for this class or interface, encoded in an integer. The modifiers consist of the Java Virtual Machine's constants for public, protected, private, final, static, abstract and interface; they should be decoded using the methods of class Modifier.
>
> If the underlying class is an array class, then its public, private and protected modifiers are the same as those of its component type. If this Class represents a primitive type or void, its public modifier is always true, and its protected and private modifiers are always false. If this object represents an array class, a primitive type or void, then its final modifier is always true and its interface modifier is always false. The values of its other modifiers are not determined by this specification.
>
> The modifier encodings are defined in The Java Virtual Machine Specification, table 4.1.

Notice that in this example, there is a description for a most common case (classes and interfaces), and then separate clarifying assertions for specific cases like arrays or primitive types.

Based on this specification, the following assertions can be identified:

Assertion	Meaning
A1	Returns the Java language modifiers for this class or interface, encoded in an integer.
A2	If the underlying class is an array class, then its public, private and protected modifiers are the same as those of its component type.
A3	If this Class represents a primitive type or void, its public modifier is always true, and its protected and private modifiers are always false.
A4	If this object represents an array class, a primitive type or void, then its final modifier is always true and its interface modifier is always false.

Now we use equivalence class partitioning and boundary value analysis techniques to write the actual tests.

The method under test has no parameters, so we can identify equivalence classes based only on the type of the object under test and on the resulting output.

Here are the equivalence classes based on the object type:

Equivalence Class	Object Type
EC1-1.	Class or interface.
EC1-2.	Array.
EC1-3.	Primitive type or void.

Here are the equivalence classes based on the returned result:

Equivalence Class	Returned Result
EC2-1.	Object, which is public.
EC2-2.	Object, which is protected.
EC2-3.	Object, which is package visible (neither of the above).
EC2-4.	Object, which is private.
EC2-5.	Object, which is static.
EC2-6.	Object, which is non-static.
EC2-7.	Object, which is final.
EC2-8.	Object, which is non-final.
EC2-9.	Object, which is abstract.
EC2-10.	Object, which is non-abstract.
EC2-11.	Object, which is a class.
EC2-12.	Object, which is an interface.

Boundary value analysis does not add much in this case. However, we may separate out void as a special case for the primitive type and one-dimensional array as a lower limit for multi-dimensional arrays.

Now we will build a set of objects that covers all of the above assertions at least once. Here is one of the possible results:

Object	Definition
O1.	Public non-static final non-abstract class. (EC1-1, EC2-1, EC2-6, EC2-7, EC2-10, EC2-11)
O2.	Protected static non-final class which must be inner. (EC2-2, EC2-5, EC2-8)
O3.	Private class which must be inner. (EC2-4)
O4.	Package-visible abstract class. (EC2-3, EC2-9)
O5.	Interface (EC2-12).

The following are objects added based on A2:

Object	Definition
O6.	One-dimensional array of O1.
O7.	Two-dimensional array of O2.
O8.	Two-dimensional array of O3.
O9.	Five-dimensional array of O4.

The following are objects added based on A3:

Object	Definition
O10.	Primitive type.
O11.	Void.

Now we can write 11 test cases for each of the above objects. Note that in some cases, such as O1, we may use J2SE APIs, and in others we have to write our own sample classes.

The resulting test code for `GetModifiersTest.java` is shown in CODE EXAMPLE 3

CODE EXAMPLE 3 GetModifiersTest.java

```
import java.lang.reflect.Modifier;

public class GetModifiersTest {

    public static void main(String[] args) throws Throwable {
        GetModifiersTest t = new GetModifiersTest();
        System.out.println("TestCase01: " + t.testCase01());
        System.out.println("TestCase02: " + t.testCase02());
        System.out.println("TestCase03: " + t.testCase03());
        System.out.println("TestCase04: " + t.testCase04());
        System.out.println("TestCase05: " + t.testCase05());
        System.out.println("TestCase06: " + t.testCase06());
        System.out.println("TestCase07: " + t.testCase07());
        System.out.println("TestCase08: " + t.testCase08());
        System.out.println("TestCase09: " + t.testCase09());
        System.out.println("TestCase10: " + t.testCase10());
        System.out.println("TestCase11: " + t.testCase11());
    }

    private boolean testCase01() {
        int m = String.class.getModifiers();
        return
          Modifier.isPublic(m) &&      // EC2-1
          ! Modifier.isStatic(m) &&    // EC2-6
          Modifier.isFinal(m) &&       // EC2-7
          ! Modifier.isAbstract(m) &&  // EC2-10
          ! Modifier.isInterface(m);   // EC2-11
    }

    private boolean testCase02() {
        int m = Object2.class.getModifiers();
        return
          Modifier.isProtected(m) &&   // EC2-2
          Modifier.isStatic(m) &&      // EC2-5
          ! Modifier.isFinal(m);       // EC2-8
    }

    private boolean testCase03() {
        int m = Object3.class.getModifiers();
        return
          Modifier.isPrivate(m);       // EC2-4
    }

    private boolean testCase04() {
        int m = Object4.class.getModifiers();
        return
          ! Modifier.isPublic(m) &&    // EC2-3
          ! Modifier.isProtected(m) && // EC2-3
          ! Modifier.isPrivate(m) &&   // EC2-3
          Modifier.isAbstract(m);      // EC2-9
```

```
}

private boolean testCase05() {
    int m = Runnable.class.getModifiers();
    return
      Modifier.isInterface(m);     // EC2-12
}

private boolean testCase06() {
    int m = String[].class.getModifiers();
    return
      Modifier.isPublic(m) &&      // A2
      Modifier.isFinal(m) &&       // A4
      ! Modifier.isInterface(m);   // A4
}

private boolean testCase07() {
    int m = Object2[][].class.getModifiers();
    return
      Modifier.isProtected(m) &&   // A2
      Modifier.isFinal(m) &&       // A4
      ! Modifier.isInterface(m);   // A4
}

private boolean testCase08() {
    int m = Object3[][].class.getModifiers();
    return
      Modifier.isPrivate(m) &&     // A2
      Modifier.isFinal(m) &&       // A4
      ! Modifier.isInterface(m);   // A4
}

private boolean testCase09() {
    int m = Object4[][][][][].class.getModifiers();
    return
      ! Modifier.isPublic(m) &&
      ! Modifier.isProtected(m) &&
      ! Modifier.isPrivate(m) &&   // A2
      Modifier.isFinal(m) &&       // A4
      ! Modifier.isInterface(m);   // A4
}

private boolean testCase10() {
    int m = Integer.TYPE.getModifiers();
    return
      Modifier.isPublic(m) &&
      ! Modifier.isProtected(m) &&
      ! Modifier.isPrivate(m) &&   // A3
      Modifier.isFinal(m) &&       // A4
      ! Modifier.isInterface(m);   // A4
}
```

```
    private boolean testCase11() {
        int m = Void.TYPE.getModifiers();
        return
          Modifier.isPublic(m) &&
          ! Modifier.isProtected(m) &&
          ! Modifier.isPrivate(m) &&   // A3
          Modifier.isFinal(m) &&       // A4
          ! Modifier.isInterface(m);   // A4
    }


    protected static class Object2 extends Exception {
    }

    private class Object3 {
        int i;
    }

}

abstract class Object4 {
}
```

Common Errors in Writing TCK Tests.

The following sections show some common errors to avoid when writing TCK tests.

Common Error: Use of Platform-Specific Data

This example examines the pitfalls of using parameter values that are specific to a particular file system or several file systems. It reads from a file named “a” in a directory whose name is calculated.

The following code will work only within file systems similar to a Microsoft Windows operating environment:

```
...
String fname = getDirName() + "\\" + "a";
FileInputStream fis = new FileInputStream(fname);
...
```

This code is more cross-platform compatible:

```
...
String fname = getDirName() + File.separatorChar + "a";
FileInputStream fis = new FileInputStream(fname);
...
```

However, even the previous code may not work on all file systems. The following code is the best option because it uses a `File` constructor to specify the parent directory:

```
...
File file = new File (getDirName(), "a");
FileInputStream fis = new FileInputStream(file);
...
```

Common Error: Modification of the System State

The following test case code changes the system default time zone in order to verify the functionality of the `TimeZone.setDefault` method. Permanent change of the default time zone may affect the execution of other tests:

```
...

// change the default time zone
    TimeZone zone = TimeZone.getTimeZone("GMT");
    TimeZone.setDefault(zone);

// verify setting
    TimeZone tz = TimeZone.getDefault();
    if (!tz.equals(zone)) {

       // test fails
       ...
    }

...
```

The correct way to write the above code is to restore the original time zone setting:

```
...

    TimeZone defaultTimeZone = TimeZone.getDefault();

    // change the default time zone
    TimeZone zone = TimeZone.getTimeZone("GMT");
    TimeZone.setDefault(zone);

    TimeZone tz = TimeZone.getDefault();

    // restore the original default time zone
    TimeZone.setDefault(defaultTimeZone);

    // verify setting
    if (!tz.equals(zone)) {
       // test fails
       ...
    }

...
```

Common Error: Stress Tests

TCK tests should typically avoid stress-testing of the API. Stress tests verify correctness of API functionality by processing a large amount of data or using most of the available shared resource. Such tests may not work correctly on all implementations, especially if resource availability on a particular implementation is limited.

It is important to differentiate stress-testing from that of boundary-value testing. Boundary value tests verify the functionality on the border of the specification, while stress tests try to push the implementation close to unspecified limits.

Examples of typical stress tests to avoid include the following:

- Opening an unreasonable amount of AWT windows
- Opening an unreasonable amount of files
- Making an unreasonable amount of socket connections
- Launching an unreasonable number of threads.
- Allocating an unreasonable amount of memory
- Using an unreasonable amount of processor time, such as by calling the method under test 10,000 times in a loop.

Note – It is not always easy to tell when the amount of a resource allocation becomes unreasonable. The rule of thumb is not to use more of a resource than is necessary for a test. A good test should not use more than just a couple of objects, windows, threads, loop cycles, or bytes of allocated memory, unless it is explicitly required or allowed by the API specification.

Common Error: Hard-Coded System-Specific Values

It is inappropriate for a test to contain data which is specific to a particular host or system. For example, the following piece of code will not work on any system:

```
...
   // open URL connection

   URL url = new URL("http://java.sun.com/index.html");
  HttpURLConnection conn = (HttpURLConnection)url.openConnection();
...
```

The correct way to write the test is to make the hard-coded value configurable by the user:

```
...
   String http_url = ... // Get the URL string
   ...

   // open URL connection

   URL url = new URL(http_url);
  HttpURLConnection conn = (HttpURLConnection)url.openConnection();
...
```

Common Error: Thread Synchronization

TCK tests should be written very carefully if they involve multi-threaded tests. For example, the `Thread.sleep()` method should never be used for thread synchronization, but rather, the wait/notify method pair should be used instead.

See this reference book for more information on multi-threaded design:

Douglas Lea; Concurrent Programming in Java, Second Edition: Design Principles and Patterns; Addison-Wesley; ISBN: 0201310090

Special Class and Method Testing Issues

The following objects introduce some special consideration when developing API tests:

- Exception classes
- Abstract classes
- Interfaces
- Inherited methods
- API Signatures

Testing Exception Classes

Java APIs make heavy use of the throwable exception handling system with many exception and error classes derived from the `java.lang.Throwable` class.

Testing requirements for new exception classes can be separated into two general types which are summarized like this:

TABLE 5 Types of Testing Requirements

Type of Exception Class	Testing Required
Adds features to the base class, such as `java.sql.SQLException`	Full API testing is required in accordance with guidelines
Only adds new exception or error names to the API, such as `java.lang.Exception`	Testing must only verify correct constructor behavior and inheritance relationships

See the sampleTCK tests for `GetQuoteException` in the Java CTT at:

CTT_HOME`/examples/sampleTCK/tck/tests/api/com_sun/tdk/sampleapi/GetQuoteException/index.html`

Testing Abstract Classes

All existing abstract classes defined by an API can satisfy one or more of the conditions below. For each condition that is true for a particular abstract class, the testing related to this condition should be performed.

TABLE 6 Abstract Class Conditions

Condition	Testing Required
There are public derived concrete implementations (non-abstract classes) available in the public API specification, such as `java.io.FilterInputStream` that extends `java.io.InputStream`.	All of the public concrete derived classes must be tested.
There are private implementations that are accessible through the public API, such as `java.text.NumberFormat.getInstance()` returning the `java.text.NumberFormat` implementation.	All of the private concrete derived classes must be tested (subject to equivalence class partitioning).
There are non-abstract methods in this abstract class, such as `java.util.AbstractList` non-abstract methods.	Use a stub class. See "Testing Using a Stub Class" on page 47.

Testing Using a Stub Class

If an abstract class has some non-abstract method, then a testing technique that uses a *stub* class is recommended regardless of the type of abstract class. The TCK should provide its own implementation of this abstract class in the form of the stub class implementation, and it is acceptable for this stub class to have little or no functionality. The behavior of all of the related non-abstract methods can then be tested using the stub class.

See sampleTCK tests for `StockSymbol` in the Java CTT at:

CTT_HOME`/examples/sampleTCK/tck/tests/api/com_sun/tdk/sampleapi/StockSymbol/index.html`

Testing Interfaces

All existing interfaces defined by an API can satisfy one or more of the conditions below. For each condition that is true for a particular interface, the testing related to this condition should be performed.

TABLE 7 Interface Conditions

Condition	Testing Required
There are public implementations available in the public API specification, such as `java.util.StreamTokenizer` that `implements java.util.Enumeration`.	All of the public implementations must be tested.
There are private implementations that are accessible through the public API, such as returning the `java.util.ListIterator` implementation.	All of the private implementations must be tested (subject to equivalence class partitioning).

If the interface in question is designated as something to be implemented by user applications (for example, java.util.EventListener), testing this interface is not required. Note that a TCK test developer may still need to provide implementations of such interfaces to test other APIs that require those interfaces as input parameters.

See sampleTCK tests for `QuoteAgent` in the Java CTT at:

CTT_HOME`/examples/sampleTCK/tck/tests/api/com_sun/tdk/sampleapi/QuoteAgent/index.html`

Testing Inherited Methods

The requirements for testing inherited methods will depend on whether the methods override the base class methods, and if so, how they are overridden, as follows:

TABLE 8 Requirements for Testing Inherited Methods

Type of Inherited Method	Testing Required
Class being tested overrides methods from its base class	Overridden methods must be re-tested in this class Note: the tests generated for the base class version of the method being tested may not be complete enough for the overriding version of the method
Methods that are not overridden in the class being tested	These *possibly* need to be re-tested. If no need to test is verified by a thorough and documented risk assessment, the test development team may forego re-testing inherited methods that have not been overridden.

See sampleTCK tests for `Quote.toString()` in the Java CTT at:

CTT_HOME`/examples/sampleTCK/tck/tests/api/com_sun/tdk/sampleapi/Quote/index.html#toString()`

Testing API Signatures

In general, writing test cases that test API signatures involves checking that an implementation provides all of the specified API in a compatible way. The test cases are meant to guarantee that if an application links without any errors with one compatible implementation of the API, it will not have any linkage problems with any other compatible implementation of the API. This is commonly referred to as signature testing.

To provide such compatibility, all implementations should be two-way binary compatible as specified in Chapter 13 of *The Java™ Language Specification*. This is the essence of what should be checked by signature testing. In particular, signature testing checks all public and protected classes and members, and does not take into account classes and members with private and package access rights.

Note that many API specifications do not allow for an implementation to subset or superset the specified API. In such cases, testing API signatures requires strict comparison of the implemented API with the requirements of the specification. All the classes and members that are required by the API under test must be provided, and no new publicly available classes and members may be supported.

In cases when an API specification permits variation of the implemented API, the API signature testing should check the following:

- The availability and signature compatibility of required classes / members
- The signature compatibility of optional classes / members when they are supported by the implementation under test

The Java CTT distribution provides the Signature Test Tool to assist in developing a signature test for inclusion in a finished TCK. See its user's guide included in the Java CTT distribution.

Test Writing Exercises

As optional exercises you may now write test cases for the following API method specifications. When you are finished with the exercises, put the work aside for continuation, because after completing Chapter 6, "Writing Tests for Execution by a Test Harness" you can adapt your test case code for integration with the JavaTest harness.

Exercise 1: `java.lang.Integer`

```
public static String toHexString(int i)
```

Creates a string representation of the integer argument as an unsigned integer in base 16.

The unsigned integer value is the argument plus 2^32 if the argument is negative; otherwise, it is equal to the argument. This value is converted to a string of ASCII digits in hexadecimal (base 16) with no extra leading 0s. If the unsigned magnitude is zero, it is represented by a single zero character '0' ('\u0030'); otherwise, the first character of the representation of the unsigned magnitude will not be the zero character.

The following characters are used as hexadecimal digits:

0123456789abcdef

Exercise 2: `java.lang.Class`

```
public Class getSuperclass()
```

Returns the Class representing the superclass of the entity (class, interface, primitive type or void) represented by this Class. If this Class represents either the Object class, an interface, a primitive type, or void, then null is returned. If this object represents an array class then the Class object representing the Object class is returned.

CHAPTER 6

Writing Tests for Execution by a Test Harness

The previous chapter described how to write Java API compatibility test code without the use of any test harness library classes. This made the examples independent of any particular test harness. In order to incorporate the test code examples previously described into an actual TCK, it is necessary to take the test harness into consideration. The test harness is defined as a collection of applications and tools that are used for test execution and test suite management.

The JCP program does not require the use of any specific test harness in a TCK being developed by an expert group. At Sun, TCK tests are executed using the JavaTest harness tools, with each TCK version running under a specified JavaTest harness version.

This chapter incorporates the use of the JavaTest harness library classes into the previous test code examples. This shows one way of writing tests to be executed by a test harness.

JavaTest Harness

The JavaTest harness is a powerful set of tools for executing tests on a variety of platforms. It can run many types of tests on a variety of Java technology implementations. You can browse results on-line within the JavaTest harness itself, or off-line in the HTML reports that the JavaTest harness generates.

To run the JavaTest harness, you specify what tests to run, how to run them, and where to put the results. The JavaTest harness graphical user interface (GUI) aids you in setting up a test run, monitoring the execution of the tests, and browsing the results.

Some TCKs include a configuration wizard / editor to assist users in configuring the TCK to the particular implementation under test. The JavaTest harness 3.0 provides a configuration wizard / editor engine and API.

In the absence of a configuration wizard, some TCKs include sample configuration files such as the JavaTest harness parameter files (*.jtp) or environment files (*.jte). These are plain text files that are manually edited by the TCK user. Experience has shown that sample or template configuration files are more difficult to support and more prone to user error than using configuration wizards.

The compatibility tests that make up the TCK compatibility test suite are precompiled and indexed within the TCK test directory structure. When a test run is started, the JavaTest harness scans through the set of tests that are located under the directories selected in the `initialFiles` parameter. While scanning, the JavaTest harness selects the appropriate tests according to certain rules and queues them up for execution. This is described further in "How Tests Are Selected for a Test Run" on page 53.

JavaTest Harness Agent

A JavaTest harness agent is a small Java application that is used in conjunction with the JavaTest harness to run tests on a Java technology implementation on which it is not possible or desirable to run the main JavaTest harness application. The agent runs on a target device where the Java technology implementation is running. When using the agent as part of a TCK to test an implementation, the expectation is that the implementation of the technology is running on a target device which is connected in some manner to a PC or a workstation. The PC or workstation hosts the JavaTest harness. This is illustrated in FIGURE 2.

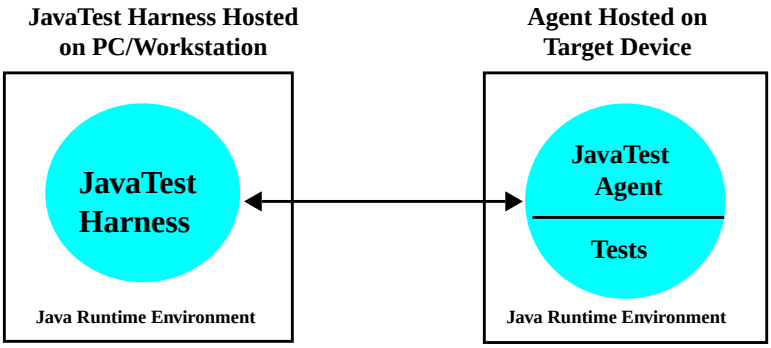


FIGURE 2 JavaTest harness with the agent hosted on a target device

The agent's responsibility on the target device is to run the tests selected by the JavaTest harness. It returns the results of these tests to the JavaTest harness by way of the particular implementation-specific communications route. The agent communications protocol has been ported to support the TCP/IP and RS-232 serial line protocols. Other stream oriented connection types are easily supported by the JavaTest Agent protocol.

Under the Harness/Agent arrangement shown in FIGURE 2 on page 52, the JavaTest harness application running on the PC or workstation performs the following functions:

1. Collects tests that are to be run.
2. Issues commands to an agent to run particular tests within the technology being tested.
3. Receives the test results from the agent and writes them to the various report files.

Note – The type of agent application(s) used by a particular TCK are determined by the TCK architect according to the characteristics of the Java technology being tested.

How Tests Are Executed by the Harness

This section describes how TCK tests are selected for a test run when using the JavaTest harness, and how they are then executed and reported on as test results. It is included only as an example. TCKs might vary from these basic test selection principles according to the functionality of the particular implementation.

How Tests Are Selected for a Test Run

Immediately prior to the start of a test run, the JavaTest harness selects tests for the run based on the following test selection criteria:

Initial files	The JavaTest harness finds tests listed in the “initial files” field of the JavaTest Harness Parameter Editor. You can specify sub-branches of the tree in the “initial files” field as a way of limiting which tests are executed during a test run. The JavaTest harness walks the tree starting with the sub-branches or tests you specify and executes all tests that it finds.
Exclude list	Tests listed in the appropriate exclude list are “deselected” prior to the start of a test run.
Prior Status	The set of tests may be optionally restricted according to the outcome of the test on a previous run. The outcome is read from the result file in the work directory that would be written if the test were to be run again.
Keywords	A test can be selected based on keywords specified in the keywords field in the test description.

`selectIf` field

The `selectIf` field contains an expression composed of elements from the test environment along with the standard operators:

```
+, -, *, /, <, >, <=, >=, &, |, !, !=, ==
```

Example:

```
integerValue>=4 & display==”my_computer:0”
```

These expressions are evaluated prior to the start of the test run. If the expression evaluates to *true*, then the test is selected as part of the test run. If the expression evaluates to *false*, or if any of the elements are not defined in the test environment, the test is not selected.

How Tests Are Executed

This section illustrates the TCK test execution model by describing how run-time tests are executed in the Java Compatibility Kit. Run-time tests are all tests that contain the `runtime` keyword in the test description.

Test Execution Steps Managed by the JavaTest Harness:

The following test execution steps are also illustrated in FIGURE 3 on page 56.

1. If the applicable test description data that is passed to the JavaTest harness contains `remote` fields, the test is recognized as a distributed test, in which case:

   a. A message switch is started to enable communication between the components running remotely. All communication between these distributed components goes through the host running the JavaTest harness.

   b. Each of the remote entries is activated

2. The main class is executed (both distributed and non-distributed tests). If execution results in an error, the test is in *error.* An error indicates that the test code was run incorrectly and it returned an unintended result—as opposed to a *failure* which could be an intended result (see Step 4 below).

3. If the test is a distributed test, the message switch is closed down, and the results from the remote components are collected and combined with the result of the main class. The final status is the “first worst” status: *failed* is worse than *pass*, *error* is worse than *failed*.

4. Positive / negative check.

   - If the test description contains the `negative` keyword the test:

     Passes if execution fails (Step 2). There are very few negative run-time tests; a few are required to test invalid `main(...)` signatures.

Fails if execution succeeds (Step 2).

- If the test has a `positive` keyword (the common case) the result of the test is the same as the result of the execution (Step 2).

Test Results

Test execution results are reported as one of the following three states:

Pass	A test passes when the functionality being tested behaves as expected. *All tests are expected to pass.*
Fail	A test fails when the functionality being tested does not behave as expected.
Error	A test is considered to be in error when something (usually a configuration problem) keeps the test from being executed as expected. Errors often indicate a systemic problem and a single configuration problem can cause many tests to fail. For example, if the path to the Java runtime environment is configured incorrectly, no tests can run and all will be in error.

The following flow chart illustrates the execution model.

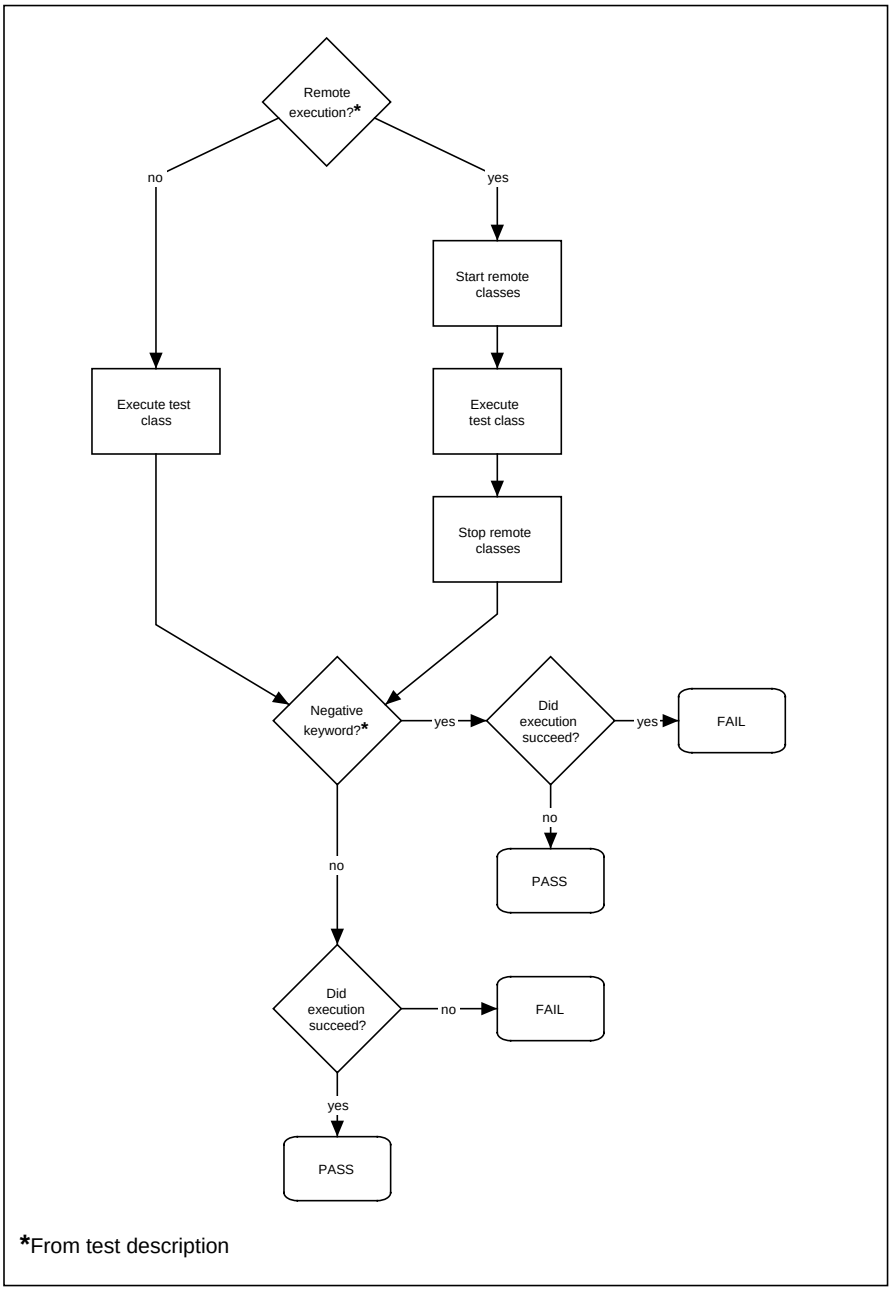


FIGURE 3 Run-time test flow diagram

Test Components Required by the JavaTest Harness

Once a test case is written into source code for eventual execution, it must be described to the test execution system. This is done so that the harness can correctly run the test. Each test harness used by a TCK has its own execution requirements reflected in the test source code. In order to illustrate a method of accomplishing this, this section describes the test components that the JavaTest harness uses to describe and define the tests for its test execution engine.

When using the JavaTest harness, the source code components included in the test suite project build work space are as follows:

- Test description: created as JavaTest harness readable HTML tables or Javadoc tag comments. They provide the data necessary to run a test.
- Test source code: executes the test case and returns results to the JavaTest harness. Includes the test case source code as described in Chapter 5, "Writing Java API Compatibility Tests" along with the JavaTest API calls, described later.
- Test case specification: HTML comments documenting what a test does and what results are expected. These comments are for test documentation purposes only, and are not used directly by the JavaTest harness for test execution.
- Other: any data files, new configuration information, or new build information that is required by a test

The following sections describe how to write test descriptions and use the API test libraries that are provided by the JavaTest harness.

Writing Test Descriptions for the JavaTest Harness

The JavaTest harness requires that each test be accompanied by machine readable descriptive data, essentially in the form of a set of test suite specific name/value pairs in either HTML or Javadoc tags. In JavaTest harness terminology, this information is called the *test description*. The JavaTest harness parses the information provided in the test description and uses it to correctly process and run tests. Whether to use HTML or Javadoc tag style for forming test descriptions is a matter of personal preference. API tests written using the JavaTest API libraries are compatible with both forms.

Running Tests with the Test Finder

The format and specific name/value pairs for the test descriptions for a particular TCK are determined by the TCK specific *test finder*. The JavaTest harness uses the test finder to locate test descriptions. A test finder is a class used to process the files of the test suite in order to locate test descriptions and run the associated tests. It reads files and finds any test descriptions in those files that describe tests to run in the test suite. The JavaTest harness provides two standard test finders named `HTMLTestFinder` and `TagTestFinder`. They correspond to the two previously described standard formats for test descriptions: HTML tables, or Javadoc tag comments, respectively. Test finders will be described in detail in the *TCK Architect's Guide*, which is scheduled for release in the near future.

Test Description Form and Content

Each test description requires a unique identifier in the form of a Uniform Resource Locator (URL). When using HTML test descriptions, the URL for the test description is the URL of the HTML document that holds the test description and, if present in the HTML document, the nearest named reference using the HTML anchor tag with a *name* attribute (detailed later in this section). When using test descriptions in the Javadoc tag style, the URL of the test description is the URL of the source code file.

Test descriptions supply the following information to the JavaTest harness:

- What source files belong to the test
- What class or executable to run
- Information to determine how the test should be run

The next section further clarifies this by describing how to create HTML tables for test descriptions.

Creating JavaTest Harness HTML Test Description Tables

Most browsers compensate for HTML pages that are syntactically incorrect. When creating HTML test description tables for the JavaTest harness it is important to remember that the JavaTest harness does not. When the JavaTest harness parses the HTML test description table, it requires syntactically correct HTML, especially in the *table* and *anchor* tags. It is best to use HTML checking tools to verify that the HTML developed for a test suite is correct.

While reading the remainder of this section, you might also refer to Appendix B, "HTML Test Description Code Listings" for detailed examples.

The JavaTest harness uses a feature of the HTML 4.0 specification called the *class* attribute. We strongly recommend specifying either the HTML 4.0 transitional or strict document type directive (DTD) in every HTML file within a test suite. Respective examples are as follows:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
```

or

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
```

Earlier test suites used a special hybrid HTML DTD developed under HTML 3.x. Most have been updated to use either the transitional or strict document type specified under HTML 4.0.

The following name/value pairs form the basis of an HTML test description table for use within a TCK test suite (a browser display of the actual HTML would look much the same).

TABLE 9 Example HTML Test Description Pairs

Name Field	Value Field
title	String Values Tests
source	StringValues.java
executeClass	javasoft.sqe.tests.api.java.lang.Integer.StringValues
keywords	runtime positive

The name field descriptions used in the table are as follows:

- *title* field describes the test in an easy to read format to be used in reports.
- *source* field lists the source code files used by the test.
- *executeClass* field lists the fully qualified name of the class to be executed to run the test.
- *keywords* field describes the logical attributes of the test which the JavaTest harness then uses to determine how to execute the test.

The actual HTML code that creates this test description would look something like this:

```
<table border=1 class=TestDescription>
 <tr><th>title</th> <td>String Values Tests</td></tr>
 <tr><th>source</th> <td>StringValues.java</td></tr>
 <tr><th>executeClass</th>
    <td>javasoft.sqe.tests.api.java.lang.Integer.StringValues</td></tr>
 <tr><th>keywords</th> <td>runtime positive</td></tr>
</table>
```

It is necessary to specify the `class` attribute for test description tables. The JavaTest harness uses this table attribute to determine which tables must be parsed as test descriptions and which tables can be ignored. The class attribute is specified within a table tag as follows (also shown in the previous HTML code sample):

```
<table ... other table attributes ... class=TestDescription>
```

As previously noted, each test description must have a unique identifier. The JavaTest harness uses a URL that is relative to the test suite root directory. There are two ways to ensure a unique URL for each test description in an HTML file. You can either:

- Place each test description into a separate HTML file, or
- Specify a named reference in the HTML file for each test description in the table by using a *name* attribute within an HTML anchor tag.

A named reference using the HTML anchor tag with a `name` attribute is formed like this:

```
<a name="myName"></a>
```

Because of the restrictions in the format of the JavaTest harness exclude lists, the value of any named reference may not have any spaces in it. The JavaTest harness uses the nearest anchor `name` tag that appears before the test description table as the URL for the test description. Descriptive information can be placed between the named reference and the test description table, as follows:

```
<a name="myName"></a>

... Some text appearing between the reference and the test
   description table ...

<table border=1 class=TestDescription>

... The rest of the test description table ...

</table>
```

If the previously described HTML document had the name `testDescrs.html` and were located in the `tests/api/foo/bar` directory of the TCK root directory, then the URL of the test description would be as follows:

```
api/foo/bar/testDescrs.html#myName
```

Test Description Field Examples

TABLE 10 contains the test description fields currently defined in the `JCKFinder` and `JCKScript` classes used by the TCK for the Java 2 Platform, Standard Edition release. The test suite you are working on may not use all of these test description fields.

TABLE 10 Example Test Description Fields

Field	Description
title	A descriptive string that identifies what the test does. The title appears in reports and in the JavaTest harness status window.
source	For runtime tests, the source field contains the names of the files previously compiled to create the test's class files. Precompiled class files are included with the TCK, and source files are included for reference only. Source files are most often `.java` files.
keywords	String tokens that can be associated with a given test. They describe attributes or characteristics of the test. Keywords are often used to select / deselect tests from a test run. Keywords are also used to select how the test will be executed by the JavaTest harness.
executeClass	The name of the class that the JavaTest harness loads and runs. This class may in turn load other classes when the test is run.
executeNative	The name of the platform-native program used to execute this test. This is specific to Java Native Interface (JNI) tests that launch a Java virtual machine from native C code.
executeArgs	An array of strings that are passed to the test classes being executed. The arguments may be fixed but often involve symbolic values that are substituted from the test environment (variables defined elsewhere in the test environment). These arguments form the basis for the set of arguments that are passed into the tests defined in the executeClass and executeNative fields. Note: If any of these values are not defined in the test environment they are passed to the test as an empty string.
rmicClasses	For RMI tests this nominates the set of classes that are passed to the RMI compiler.
timeout	A value specified in seconds used to override the default time-out used for TCK tests.

TABLE 10 Example Test Description Fields

Field *(Continued)*	Description
context	Specifies configuration values required by the test. The JavaTest harness checks to be sure these values are set before it runs the test and then passes the values through to the test. If any of these values are not defined, the JavaTest harness reports an error.
selectIf	Specifies a condition that must be satisfied in order for the test to be executed. The selectIf field makes it possible to execute (or not execute) tests based on values in the test environment. This field is constructed using environment values and the full set of boolean operators (!, <, <=, >, >=, ==, !=, &, |). The following is an example: `IntegerValue>=4 & display=="my_computer:0"` If the boolean expression evaluates to false the test is not run. If the expression evaluates to true the test is run. If any of the values are not defined in the test environment, the JavaTest harness considers the test to be in error.

JavaTest Validation of Test Descriptions

Each test suite can contain both required and optional test description files for its tests. You should review the test suite documentation to determine how to use them when developing tests. The JavaTest harness validates the following test description features while running:

- Each test description has a unique URL.
- Each anchor name tag used as a test description URL has no spaces.
- Any hypertext links are valid.
- All required test description fields are present.
- No obsolete or unknown test description fields are present.
- No obsolete or unknown keywords are present in the keywords field.

Using Keywords in Test Descriptions

Keywords are tokens associated with specific tests and they have two functions:

- To convey information to the JavaTest harness about how to execute the tests
- To serve as a basis for including and excluding tests during test runs

Keyword expressions are normally specified in the *keywords* field in the JavaTest Parameter Editor or the JavaTest Configuration Editor. They filter tests during test runs according to the *keyword* field of the test description.

The architect or lead of each TCK will specify the set of valid keywords for that TCK. TABLE 11 lists some generally accepted test description keywords.

TABLE 11 Generally Accepted Test Description Keywords

Keyword	Meaning
interactive	Identifies tests that require human interaction.
negative	The component under test must terminate with (and detect) an error. Another way to describe a negative test is that the test must fail on the component under test in order for it to be deemed to have passed.
positive	The component under test must terminate normally. Another way to describe a positive test is that the test must succeed on the component under test in order to have passed.
runtime	Identifies tests used with Java run-time products.

Using the JavaTest API Test Libraries

The JavaTest harness provides several levels of API support for test developers. The most basic level is represented by the `Test` interface and the `Status` class, as follows:

- `Test` interface – used as a test entry point, named `com.sun.javatest.Test`.
- `Status` class – used as a convenient means of reporting the outcome of tests, named `com.sun.javatest.Status`.

There are also library classes that provide additional services to the tests. These services may range from generic tasks like grouping several test cases together into one .java file, to tasks which are specific to a particular category of tests. This chapter will cover only the most generic of these classes, which is the class `MultiTest`, fully named `com.sun.javatest.lib.MultiTest`.

The `Test` Interface

This is the interface that all API tests must implement. It has only one `run()` method, as follows:

```
public Status run(String[] args, PrintWriter log, PrintWriter ref)
```

Using this method, each test is provided with a set of string arguments and two writers. The exact set of arguments passed to the test is determined by the values in the test description table. Note that the argument-related fields of the test description table may contain variables in it. In this case all variables are substituted with their actual values by the JavaTest harness.

Tests may use the two writers to report messages and errors, and also to write to a reference output. The test returns its outcome through the `Status` object, which will now be described.

Using the `Status` Class

The `Status` class represents a wrapper for both the integer status code of a test and a related message. The most often used methods in this wrapper class are two static methods which offer a convenient means of improving test readability, as follows:

public static Status passed (String)

public static Status failed (String)

A typical example of using these methods is as follows:

```
if (<some test situation is checked>) {
    return Status.passed("OKAY");
} else {
    log.println(<the details of the test situation>);
    return Status.failed(<test situation name>);
}
```

Note the use of the `log` object, which is an object of type `PrintWriter` that is specified as an argument to the `run()` method.

Using the `MultiTest` Class.

The main functions of the `MultiTest` library class are as follows:

- Facilitate combining several test cases into one `.java` file.
- Provide a means of parsing arguments.
- Allow the definition of certain setup and cleanup routines for a set of tests.

Test developers typically extend `MultiTest` and define several test cases within the new class body. Each test case uses a specific test case method of the following form:

```
public Status Test_Case_Method_Name()
```

A basic example of using `MultiTest` in this fashion is as follows:

```
public class MyTest extends MultiTest {

  public Status testCase1() {     ....   }
  public Status testCase2() {     ....   }

  ....

  public Status testCaseN() {     ....   }
}
```

This example is actually quite sufficient for a simple series of tests that do not require any setup and do not use any arguments. `MultiTest` uses the reflection API to first select and then call all of the methods implemented in the extended class.

There are other JavaTest API library classes that extend `MultiTest` to be used for a specific testing purpose. An example is class `InteractiveTest` which is designed to interactively test AWT functionality. These classes and their functionality are described in detail in the JavaTest API documentation.

Integrating Test Case Code with the JavaTest API

This section continues with the test case code examples introduced in Chapter 5, "Writing Java API Compatibility Tests" and demonstrates how they are integrated into the test harness using the JavaTest API. Studying these examples will help clarify the previous description of how to integrate test case code with the JavaTest API.

Example 1: Integrating the `Integer.toString(int, int)` Test with the JavaTest Harness

The basic test case code for this example was previously described in "Example 1: TCK Tests for `Integer.toString(int, int)`" on page 33. This example builds on that test case code by showing the two files that are required to run the test case when integrated with the JavaTest harness. These two files are as follows:

- `ToStingTests.java`
  Contains the test case source code using the JavaTest API calls, this file is shown in CODE EXAMPLE 4 on page 66.

- `toString.html`
  Contains the test specifications and the test description table, this file is shown in FIGURE 4 on page 68.

Note that in this example it is not absolutely necessary to use the `MultiTest` class since there is only one test case. However, even with only one basic test case it is better to use `MultiTest` instead of just the `Test` interface because the code is actually more readable and simpler to write. For this reason, it is recommended to always use `MultiTest` for running tests.

CODE EXAMPLE 4 Source code file for `ToStingTests.java`

```
/*
 *
 * Copyright (c) 2001 Sun Microsystems, Inc.  All Rights Reserved.
 *
 * Tests for public static String toHexString(int i)
 */

package javasoft.sqe.tests.api.java.lang.Integer;

import java.io.PrintWriter;
import com.sun.javatest.Status;
import com.sun.javatest.lib.MultiTest;


public class ToStringTests extends MultiTest {

    private static String myToString(int i, int radix) {
        char[] digits = {
                '0' , '1' , '2' , '3' , '4' , '5' ,
                '6' , '7' , '8' , '9' , 'a' , 'b' ,
                'c' , 'd' , 'e' , 'f' , 'g' , 'h' ,
                'i' , 'j' , 'k' , 'l' , 'm' , 'n' ,
                'o' , 'p' , 'q' , 'r' , 's' , 't' ,
                'u' , 'v' , 'w' , 'x' , 'y' , 'z'};

        if (radix < Character.MIN_RADIX || radix >
   Character.MAX_RADIX) {
            radix = 10;
        }
      char buf[] = new char[65];
        boolean negative = (i < 0);
        int charPos = 64;

        if (!negative) {
            i = -i;
       }
```

```
        while (i <= -radix) {
            buf[charPos--] = digits[(int)(-(i % radix))];
          i = i / radix;
        }
        buf[charPos] = digits[(int)(-i)];

       if (negative) {
            buf[--charPos] = '-';
        }

        return new String(buf, charPos, (65 - charPos));
    }

    /* standalone interface */
    public static void main(String argv[]) {
        ToStringTests test = new ToStringTests();
        test.run(argv, System.err, System.out).exit();
    }

    public Status testToString() {
        int[] iValues = { Integer.MIN_VALUE, -1255, -1, 0,
                1, 34, Integer.MAX_VALUE
        };
        int[] radixValues = { Integer.MIN_VALUE, -7,
                Character.MIN_RADIX - 1, Character.MIN_RADIX,
                20, Character.MAX_RADIX, Character.MAX_RADIX + 1,
                179, Integer.MAX_VALUE
        };
        boolean pass = true;
        String expected, result;
        for (int i = 0; i < iValues.length; ++i) {
            for (int j = 0; j < radixValues.length; ++j) {
                expected = myToString(iValues[i], radixValues[j]);
             result = Integer.toString(iValues[i], radixValues[j]);
                if (! result.equals(expected)) {
                     pass = false;
                     ref.println("Failed for i=" + iValues[i] + ",
   radix=" +
                        radixValues[j] + "; expected: " + expected +
                             "returned: " + result);
                }
            }
        }
        if (pass) {
            return Status.passed("OKAY");
        } else {
           return Status.failed("public static String toString(int
   i, int          radix)");
        }
    }
}
```

`toString.html` Test Description File

FIGURE 4 is a reproduction of a browser rendering of the `toString.html` test description file. This file contains the test specification documentation on equivalence class partitioning and boundary value analysis. It also contains the test description table used by the JavaTest harness to run the `ToStingTests.java` test example. The actual code is listed in Appendix B in "`toString.html` Test Description Code" on page 101.

FIGURE 4 `toString.html` test description file

Test Specifications and Descriptions for Integer.toString(int, int)

---

public static String toString(int i, int radix)

---

public static String toString(int i, int radix)

Description

Domain testing of input and output conditions, and external pre-conditions for class Integer, method public static String toString(int i, int radix).

i	Radix	Expected Output	Test Case ID
negative	less than Character.MIN_RADIX or greater than Character.Max_RADIX	string representation of the i in base 16, starting with the minus sign	testToString
negative	between Character.MIN_RADIX and Character.MAX_RADIX	string representation of the i in base 'radix', starting with the minus sign	testToString
non-negative	less than Character.MIN_RADIX or greater than Character.MAX_RADIX	string representation of the i in base 16, starting without a sign	testToString
non-negative	between Character.MIN_RADIX and Character.MAX_RADIX	string representation of the i in base 'radix', starting without a sign	testToString

i	Radix	Expected Output	Test Case ID
Integer.MIN_VALUE, -1	Integer.MIN_VALUE, Character.MIN_RADIX -1, Character.MAX_RADIX + 1, Integer.MAX_VALUE	string representation of the i in base 16, starting with the minus sign	testToString
Integer.MIN_VALUE, -1	Character.MIN_RADIX, Character.MAX_RADIX	string representation of the i in base 'radix', starting with the minus sign	testToString
0, 1, Integer.MAX_VALUE	Integer.MIN_VALUE, Character.MIN_RADIX -1, Character.MAX_RADIX + 1, Integer.MAX_VALUE	string representation of the i in base 16, starting without a sign	testToString
0, 1, Integer.MAX_VALUE	Character.MIN_RADIX, Character.MAX_RADIX	string representation of the i in base 'radix', starting without a sign	testToString

Test Descriptions

Test cases included:
testToString.

title	Tests for public static String toHexString(int i)
source	ToStringTests.java
executeClass	javasoft.sqe.tests.api.java.lang.Integer.ToStringTests
keywords	runtime positive

Example 2: Integrating the `Class.getModifiers()` Test with the JavaTest Harness

The basic test case code for this example was previously described in "Example 2: TCK Tests for `Class.getModifiers()`" on page 37. This example builds on that test case code by showing the two files that are required to run the test case when integrated with the JavaTest harness. These two files are as follows:

- `GetModifiersTests .java`
  Contains the test case source code using the JavaTest API calls, shown in CODE EXAMPLE 5.
- `getModifiers.html`
  Contains the test specificatio s and the test description table, shown in FIGURE 4 on page 68.

CODE EXAMPLE 5 `GetModifiersTests.java`

```
/*
 *
 * Copyright (c) 2001 Sun Microsystems, Inc.  All Rights Reserved.
 */
package javasoft.sqe.tests.api.java.lang.Class;

import java.io.PrintWriter;
import com.sun.javatest.Status;
import com.sun.javatest.lib.MultiTest;
import java.lang.reflect.Modifier;

public class GetModifiersTests extends MultiTest {

    protected static class Object2 extends Exception {
    }

    private class Object3 {
        int i;
    }

    /* standalone interface */
    public static void main(String argv[]) {
        GetModifiersTests test = new GetModifiersTests();
        test.run(argv, System.err, System.out).exit();
    }
    /**
     * Equivalence class partitioning
     * with state and output values orientation
     * for public int getModifiers(),
     * <br><b>pre-conditions</b>: this object: a class,
     * <br><b>output</b>: modifiers with 'interface' bit unset.
     */
```

```
public Status testCase01() {
    int m = String.class.getModifiers();
    return (
            Modifier.isPublic(m) &&      // EC2-1
            ! Modifier.isStatic(m) &&    // EC2-6
            Modifier.isFinal(m) &&       // EC2-7
            ! Modifier.isAbstract(m) &&  // EC2-10
            ! Modifier.isInterface(m)    // EC2-11
            ) ? Status.passed("OKAY"):
            Status.failed("Failed");
}

/**
 * Equivalence class partitioning
 * with state and output values orientation
 * for public int getModifiers(),
 * <br><b>pre-conditions</b>: this object: non-final,
 * <br><b>output</b>: modifiers with 'final' bit unset.
 */
public Status testCase02() {
    int m = Object2.class.getModifiers();
    return (
            Modifier.isProtected(m) &&   // EC2-2
            Modifier.isStatic(m) &&      // EC2-5
            ! Modifier.isFinal(m)        // EC2-8
            ) ? Status.passed("OKAY"):
            Status.failed("Failed");
}

/**
 * Equivalence class partitioning
 * with state and output values orientation
 * for public int getModifiers(),
 * <br><b>pre-conditions</b>: this object: private,
 * <br><b>output</b>: modifiers with 'private' bit unset.
 */
public Status testCase03() {
    int m = Object3.class.getModifiers();
    return (
            Modifier.isPrivate(m)        // EC2-4
            ) ? Status.passed("OKAY"):
            Status.failed("Failed");
}

/**
 * Equivalence class partitioning
 * with state and output values orientation
 * for public int getModifiers(),
 * <br><b>pre-conditions</b>: this object: abstract,
 * <br><b>output</b>: modifiers with 'abstract' bit set.
 */
public Status testCase04() {
```

```
        int m = Object4.class.getModifiers();
        return (
                ! Modifier.isPublic(m) &&    // EC2-3
                ! Modifier.isProtected(m) && // EC2-3
                ! Modifier.isPrivate(m) &&   // EC2-3
                Modifier.isAbstract(m)       // EC2-9
                ) ? Status.passed("OKAY"):
                Status.failed("Failed");
    }

    /**
     * Equivalence class partitioning
     * with state and output values orientation
     * for public int getModifiers(),
     * <br><b>pre-conditions</b>: this object: interface,
     * <br><b>output</b>: modifiers with 'interface' bit set.
     */
    public Status testCase05() {
        int m = Runnable.class.getModifiers();
        return (
                Modifier.isInterface(m)      // EC2-12
                ) ? Status.passed("OKAY"):
                Status.failed("Failed");
    }

/**
 * Assertion testing
 * for public int getModifiers(),
 * <br><b>pre-conditions</b>: this object: array of public
    * classes,
 * If this object represents an array class, a primitive type
 * or void, then its final modifier is always true and its
 * interface modifer is always false..
 */
    public Status testCase06() {
        int m = String[].class.getModifiers();
        return (
                Modifier.isPublic(m) &&      // A2
                Modifier.isFinal(m) &&       // A4
                ! Modifier.isInterface(m)    // A4
                ) ? Status.passed("OKAY"):
                Status.failed("Failed");
    }

/**
 * Assertion testing
 * for public int getModifiers(),
 * <br><b>pre-conditions</b>: this object: array of protected
 * classes,
 * If this object represents an array class, a primitive type
 * or void, then its final modifier is always true and its
 * interface modifer is always false..
```

```
 */
    public Status testCase07() {
        int m = Object2[][].class.getModifiers();
        return (
                Modifier.isProtected(m) &&   // A2
                Modifier.isFinal(m) &&       // A4
                ! Modifier.isInterface(m)    // A4
                ) ? Status.passed("OKAY"):
                Status.failed("Failed");
    }

/**
 * Assertion testing
 * for public int getModifiers(),
 * <br><b>pre-conditions</b>: this object: array of private
 * classes,
 * If this object represents an array class, a primitive type
 * or void, then its final modifier is always true and its
 * interface modifer is always false..
 * visible classes,
 */
    public Status testCase08() {
        int m = Object3[][].class.getModifiers();
        return (
                Modifier.isPrivate(m) &&     // A2
                Modifier.isFinal(m) &&       // A4
                ! Modifier.isInterface(m)    // A4
                ) ? Status.passed("OKAY"):
                Status.failed("Failed");
    }

/**
 * Assertion testing
 * for public int getModifiers(),
 * <br><b>pre-conditions</b>: this object: array of package-
 * visible classes,
 * If this object represents an array class, a primitive type
 * or void, then its final modifier is always true and its
 * interface modifer is always false..
 */
    public Status testCase09() {
        int m = Object4[][][][][].class.getModifiers();
        return (
                ! Modifier.isPublic(m) &&
                ! Modifier.isProtected(m) &&
                ! Modifier.isPrivate(m) &&   // A2
                Modifier.isFinal(m) &&       // A4
                ! Modifier.isInterface(m)    // A4
                ) ? Status.passed("OKAY"):
                Status.failed("Failed");
    }
```

```
/**
 * Assertion testing
 * for public int getModifiers(),
 * <br><b>pre-conditions</b>: this object: primitive type,
 * If this object represents an array class, a primitive type
 * or void, then its final modifier is always true and its
 * interface modifer is always false.
 */
    public Status testCase10() {
        int m = Integer.TYPE.getModifiers();
        return (
                Modifier.isPublic(m) &&
                ! Modifier.isProtected(m) &&
                ! Modifier.isPrivate(m) &&   // A3
                Modifier.isFinal(m) &&       // A4
                ! Modifier.isInterface(m)    // A4
                ) ? Status.passed("OKAY"):
                Status.failed("Failed");
    }

/**
 * Assertion testing
 * for public int getModifiers(),
 * <br><b>pre-conditions</b>: this object: void,
 * If this object represents an array class, a primitive type
 * or void, then its final modifier is always true and its
 * interface modifer is always false..
 */
    public Status testCase11() {
        int m = Void.TYPE.getModifiers();
        return (
                Modifier.isPublic(m) &&
                ! Modifier.isProtected(m) &&
                ! Modifier.isPrivate(m) &&   // A3
                Modifier.isFinal(m) &&       // A4
                ! Modifier.isInterface(m)    // A4
                ) ? Status.passed("OKAY"):
                Status.failed("Failed");
    }
}

abstract class Object4 {
}
```

`getModifiers.html` Test Description File

FIGURE 5 is a reproduction of a browser rendering of the `getModifiers.html` test description file. This file contains the test specification documentation on equivalence class partitioning and assertion testing, as well as the test description

table used by the JavaTest harness to run the `GetModifiersTests.java` test example. The actual code is listed in Appendix B in "`getModifiers.html` Test Description Code" on page 104.

FIGURE 5 `getModifiers.html` test description file

Test Specifications and Descriptions for Class.getModifiers()

---

public int getModifiers()

---

public int getModifiers()

Description

Domain testing of input and output conditions, and external pre-conditions for class Class, method public int getModifiers().

Equivalence Class Partitioning

Pre-conditions	Expected output value	Test Case ID
this object: public	modifiers with 'public' bit set	testCase01
this object: non-static	modifiers with 'static' bit unset	testCase01
this object: final	modifiers with 'final' bit set	testCase01
this object: non-abstract	modifiers with 'abstract' bit unset	testCase01
this object: a class	modifiers with 'interface' bit unset	testCase01
this object: protected	modifiers with 'protected' bit set	testCase02
this object: static	modifiers with 'static' bit set	testCase02
this object: non-final	modifiers with 'final' bit unset	testCase02
this object: private	modifiers with 'private' bit unset	testCase03
this object: package visible	modifiers with neither of 'public', 'private' or 'protected' bits set	testCase04
this object: abstract	modifiers with 'abstract' bit set	testCase04
this object: interface	modifiers with 'interface' bit set	testCase05

Assertion testing

Pre-conditions	Assertion	Test Case ID
this object: array of public classes	If the underlying class is an array class, then its public, private and protected modifiers are the same as those of its component type.	testCase06
this object: array of public classes	If this object represents an array class, a primitive type or void, then its final modifier is always true and its interface modifier is always false.	testCase06
this object: array of protected classes	If the underlying class is an array class, then its public, private and protected modifiers are the same as those of its component type.	testCase07
this object: array of protected classes	If this object represents an array class, a primitive type or void, then its final modifier is always true and its interface modifier is always false.	testCase07
this object: array of private classes	If the underlying class is an array class, then its public, private and protected modifiers are the same as those of its component type.	testCase08
this object: array of private classes	If this object represents an array class, a primitive type or void, then its final modifier is always true and its interface modifier is always false.	testCase08
this object: array of package-visible classes	If the underlying class is an array class, then its public, private and protected modifiers are the same as those of its component type.	testCase09
this object: array of package-visible classes	If this object represents an array class, a primitive type or void, then its final modifier is always true and its interface modifier is always false.	testCase09
this object: primitive type	If this Class represents a primitive type or void, its public modifier is always true, and its protected and private modifiers are always false.	testCase10

this object: primitive type	If this object represents an array class, a primitive type or void, then its final modifier is always true and its interface modifier is always false.	testCase10
this object: void	If this Class represents a primitive type or void, its public modifier is always true, and its protected and private modifiers are always false.	testCase11
this object: void	If this object represents an array class, a primitive type or void, then its final modifier is always true and its interface modifier is always false.	testCase11

Test Descriptions

Test cases included:
testCase01, testCase02, testCase03, testCase04, testCase05, testCase06, testCase07, testCase08, testCase09, testCase10, testCase11.

title	GetModifiers
source	GetModifiersTests.java
executeClass	javasoft.sqe.tests.api.java.lang.Class.GetModifiersTests
keywords	runtime positive

Test Writing Exercises

This sections offers a continuation of the previous optional exercises.

To perform the exercises, you should now adapt the tests that you developed as exercises in "Test Writing Exercises" on page 50 for use with the JavaTest harness. When you are finished, you can compare your test writing work with Appendix A, "Test Writing Exercise Answers."

CHAPTER 7

Test Integration and Quality Assurance (QA)

This chapter describes the general processes and procedures used at Sun to integrate compatibility tests into a TCK and perform QA testing of the TCK product.

Product Testing the TCK Tests

A TCK and its included source code is a software product that is delivered to customers. Like any other product, it requires product testing to determine if it meets the project goals. There are three different aspects to product testing a TCK:

1. Testing against the reference implementation (RI)
2. Testing the integration of the TCK
3. Reviewing and inspecting the tests

Note – This chapter focuses on testing the TCK against the RI. In this scenario the QA team executes the TCK with the RI to test the quality of the TCK product itself. Testing the RI for compatibility by using the TCK is a separate testing issue.

Testing Against the Reference Implementation (RI)

Each Java technology release is composed of three interrelated components, each of which is usually developed by a separate team. These components are the specification, the reference implementation, and the TCK, as illustrated in FIGURE 6.

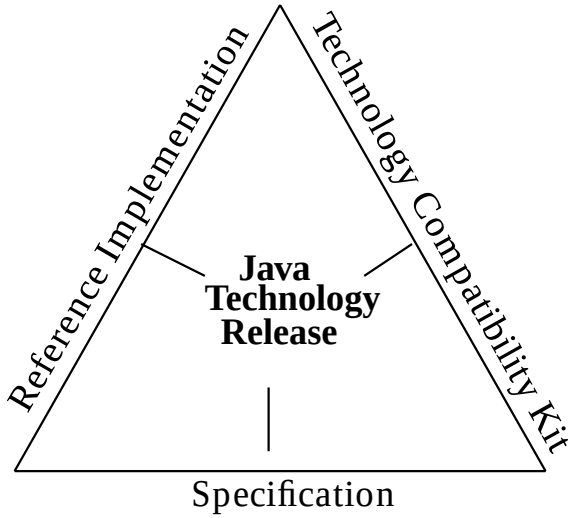
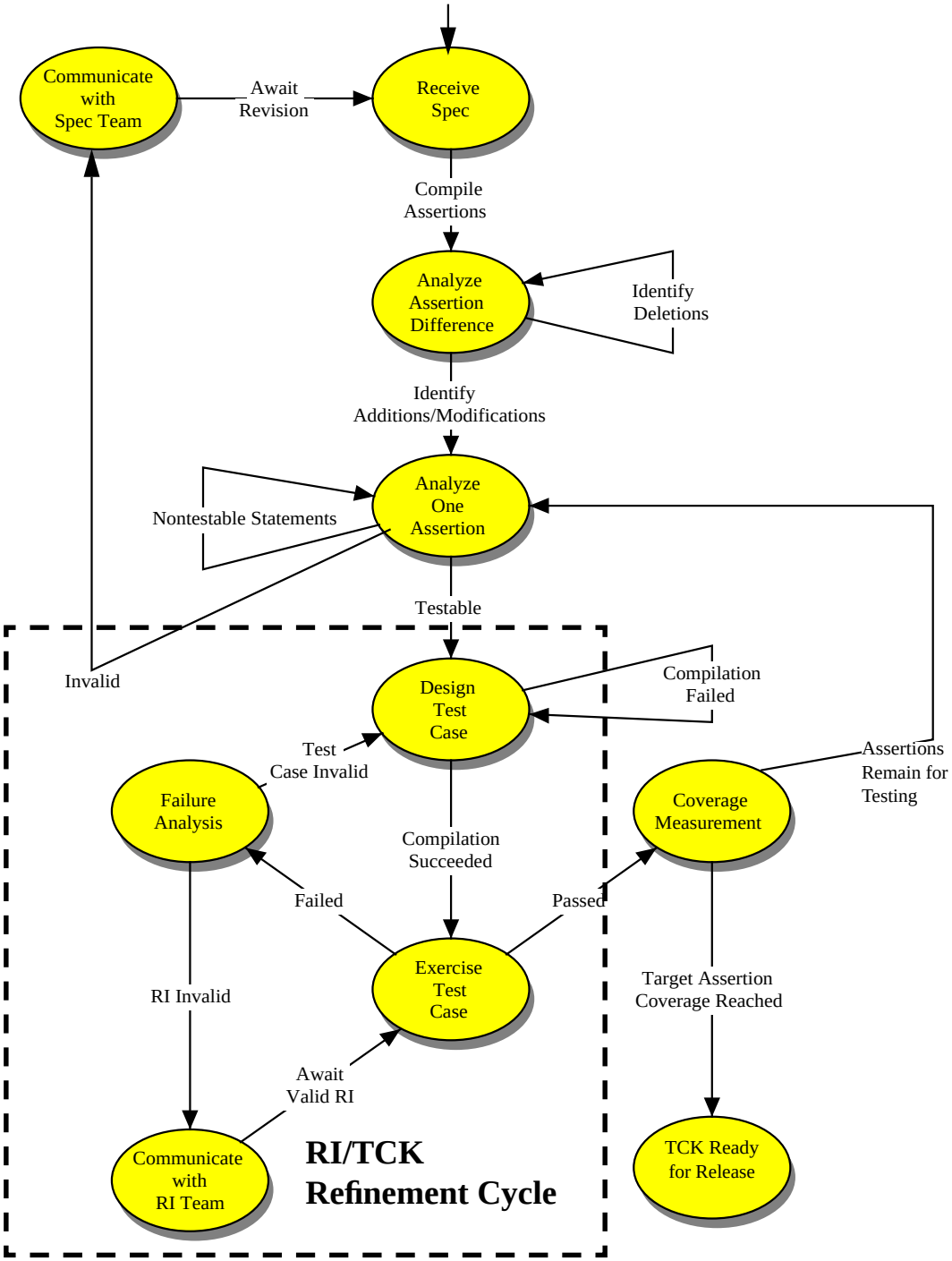


FIGURE 6 Components of a Java technology release

Each TCK is usually developed in parallel with its related RI. The TCK test development team must work closely with the RI development team in refining the RI and TCK. The goal is that the RI must pass all tests included in the finished TCK.

The process is accomplished in an iterative fashion with the RI team working closely with the compatibility test development team. This iterative process is illustrated in FIGURE 7 on page 81 within the section of the test development state flow diagram that is outlined by broken lines.

FIGURE 7 TCK / RI refinement within the test development process



Avoiding Overlap When Testing the RI and the TCK

There is often some form of overlap in product testing the RI and compatibility testing it with its related TCK. An overlap of this sort means there is duplication of effort between the product tests for the RI and the TCK compatibility tests. There should be a strategy in place to avoid this situation.

TCK tests are in a large part focused on assertion-based API tests to verify compatibility of an implementation to the specification. To avoid a duplication of effort, the RI QA team should not focus on writing assertion-based API tests.

When developing tests, the RI QA team should focus on feature testing, system testing, and stress testing. However, the team should still write supplementary API tests for any testing requirements that are not covered by the TCK tests.

Note that for the RI QA team to work efficiently in this context, it needs sufficient information about the API tests provided in the TCK. This can be accomplished by having the RI QA team review the testable assertions or the test descriptions written by the TCK test development team.

Following these suggestions helps to maximize the productivity of both teams.

Testing the Integration of the TCK

Once compatibility tests have been written, you must integrate them as source code files into a workspace that is eventually built into a finished TCK product. The use of dedicated tools greatly facilitates this process.

TCK development groups use their own choice of tools for the purpose of version control, and for making and building the finished TCK. Some of the in-house issues to be addressed are as follows:

- Managing the quality assurance of each incremental build version as the project progresses
- Implementing partial builds to test selected parts of the build process for errors (eliminating the need to perform a complete build after minor changes)
- Make utilities
- Source code version control

Integration Testing Issues

There should be documented procedures and tests to verify the correctness of the compatibility test suite during the integration process. The integration tests attempt to confirm points such as the following.

- One or more tests have passed a formal inspection.
- Source code syntax and formatting is correct, and the source code has been spell checked.
- File name constraints are observed for all platforms that the test suite can possibly be executed on.
- Tests work correctly in all applicable environments; for example, in single-process versus multi-process environments.
- All non-excluded tests run and pass on the reference implementation.
- The finished TCK package is complete with documentation, tests, classes, and tools properly in place.
- If the TCK is distributed as a file archive, such as a .zip file, the archive unpacks with the files properly in place and functioning correctly.
- Any other appropriate testing is performed.

When using the JavaTest harness, the following points are also verified by the integration tests and procedures:

- All HTML files used within the test suite tree are valid and correctly linked.
- The JavaTest harness configuration and exclude list files are correctly in place, which involves a check of the following:

  Test environment files used by the TCK (`.jte` files,)

  Configuration files required to run the tests,

  Individual exclude list entries or bug tracking items that need attention.

Reviewing and Inspecting the Tests

In addition to integration testing it is also advisable to institute a formal process for peer review and inspection of the tests. Performing formal compatibility test suite reviews and inspections at strategic points in the development process provides a powerful way to accomplish the following goals:

- Detect defects as early as possible in the compatibility test development cycle.
- Prevent the migration of defects to later phases.
- Improve test suite test coverage
- Produce fewer invalid tests.

- Improve communication between members of each of the project teams: specification, RI, and TCK.
- Provide additional peer insights on the work to the compatibility test developers.
- Improve the quality and productivity of the test development process.
- Reduce the cost and cycle time.
- Reduce the maintenance effort.

Reviews and Inspections

When to use a review process versus a more formal inspection during a project is largely a matter of degree.

Review

A *review* is defined in this context as an informal type of technical evaluation by a group of people. Someone distributes material and solicits comments. A review meeting may or may not be held. Typically there is no follow up to verify that all the issues that are raised get resolved.

Inspection

An *inspection* is a more structured and formal peer review. Its purpose is to find and eliminate any existing or eventual defects in the TCK product as it is being developed.

A moderator leads the entire inspection process, ideally, someone who is experienced and trained in the techniques being used. During the inspection process:

- Checklists and other analytical techniques are used to identify defects.
- A review meeting is held.
- Metrics are collected.
- Results are tracked.

Review vs. Inspection

Experience in the software industry has shown inspections to be the most effective type of evaluation because it is by definition more in-depth.

There is a need to strike a balance between unnecessary formality, and a structure that is sufficient to achieve success. During a project, both reviews and inspections are normally used at some point. An inspection should definitely be used when a deliverable is considered risky, for example, when it is complex or critical.

CHAPTER 8

TCK Maintenance

This chapter describes the issues surrounding TCK test maintenance after the Java technology has been released. TCK maintenance procedures are developed according to JCP program guidelines.

The TCK project plan should have an extended maintenance plan to cover the expected period of use for that TCK. Typically, a TCK is used for a period of at least two years.

TCK Anomaly Analysis

Sometimes anomalies are discovered in the TCK after its first customer shipment (FCS) release. In most cases, a TCK user reports a TCK test failure or problem which is suspected to be caused by a bug in either the TCK, the reference implementation, or the specification.

Test Appeals Process

The JCP program includes a test appeals process. The appeals process defines the escalation process in which challenges to compatibility tests are evaluated and either accepted or rejected.

As part of the documentation released with a TCK within the JCP program, the Expert Group must identify an appeals process through which challenges to the TCK can be addressed. This appeals process is managed by the Maintenance Lead (the designated expert responsible for maintaining the related technology specification).

For guidelines on establishing an appeals process, see the *TCK Project Planning and Development Guide* included in the Java CTT distribution.

Exclude List for TCK Maintenance

TCKs developed by Sun use an exclude list file to omit one or more tests from a particular test run. The appropriate the JavaTest harness exclude list file for each TCK specifies those tests that are known to be invalid, and therefore, are not required to be successful or even to be run using a valid TCK system configuration. The JavaTest harness exclude list file uses a `.jtx` extension by convention.

There are three basic reasons a test would be excluded:

- An error in the specification that was used as the basis of the test has been discovered.
- An error in the test has been discovered.
- An error in the reference implementation has been discovered after its release.

In any of these situations, the test is a candidate for exclusion. When Sun releases a TCK as the Maintenance Lead, test review and exclusion is assigned primarily to the test developer.

TCK Patches

Sun's use of the exclude list is a short term solution for problematic tests until the release of the next version of the technology with its accompanying TCK. Another short-term solution is the possibility of issuing a TCK patch. A patch is more suitable in situations where an anomaly is deemed to affect too many tests to use exclusion. In this regard, patches typically include fixes for the framework code, or the configuration files or libraries that affect a large number of tests.

An example of a patch scenario is a situation where certain tests are actually valid according to the technology specification, but they are later found to assume certain resource availability which is not explicitly granted by the specification. This might surface in a technology implementation where the tests have been found to launch an unreasonable amount of threads in an otherwise compatible implementation. This would cause problems in small, resource-constrained implementations. In this case, the sizable amount of alternative tests in question might be provided by a TCK patch.

Maintenance Releases

Exclude lists and patches may be appropriate for some TCKs, but may not be adequate for TCKs which have been in active use for several years. In this case, a maintenance release for the TCK is an alternative to a major version release.

A TCK maintenance release might include items such as the following:

- TCK bug fixes
- Specification coverage improvements, such as new tests for uncovered specification assertions that are not included in the existing TCK

Note that the following types of changes unnecessarily complicate TCK maintenance and are not recommended:

- Cosmetic changes in the test code
- Arbitrary changes to test names
- Substantial changes in the purpose or the functionality of a test without changing the test name.

TCK Evolution

A new version of the specification typically requires a new TCK version. The new TCK version normally does the following:

- Includes tests for any new specification assertions.
- Includes corrections to existing tests in cases where specification assertions have been clarified.
- Eliminates tests if they are not compatible with the new specification version.

Maintaining Multiple TCK Releases

Releases of multiple interrelated TCKs are a considerable challenge to maintain. When a change is made or a problem is found in one of the TCK releases, it is necessary to investigate how the problem affects the other interrelated releases. In these cases, any fix should be applied to all affected releases. For example, a single test failure may necessitate numerous exclude list updates.

APPENDIX A

Test Writing Exercise Answers

This appendix contains answers to the optional exercises. There are also cross-references to each of the associated HTML files that are located in Appendix B, "HTML Test Description Code Listings."

The answers included in this appendix are:

Exercise Answer: ToHexStringTests.java

CODE EXAMPLE 6 ToHexStringTests.java

```
/*
 *
 * Copyright (c) 2001 Sun Microsystems, Inc.  All Rights Reserved.
 *
 * Tests for public static String toHexString(int i)
 */

package javasoft.sqe.tests.api.java.lang.Integer;

import java.io.PrintWriter;
import com.sun.javatest.Status;
import com.sun.javatest.lib.MultiTest;


public class ToHexStringTests extends MultiTest {

    public static String myToHexString(int i) {
        char[] digits = {
                '0' , '1' , '2' , '3' , '4' , '5' ,
                '6' , '7' , '8' , '9' , 'a' , 'b' ,
                'c' , 'd' , 'e' , 'f'};

        char[] buf = new char[64];
        int charPos = 64;
        int radix = 16;
        int mask = radix - 1;

        do {
            buf[--charPos] = digits[(int)(i & mask)];
            i >>>= 4;
        } while (i != 0);

        return new String(buf, charPos, (64 - charPos));
    }

    /* standalone interface */
    public static void main(String argv[]) {
        ToHexStringTests test = new ToHexStringTests();
        test.run(argv, System.err, System.out).exit();
    }

    /**
     * Equivalence class partitioning
```

```
     * with input and output values orientation
     * for public static String toHexString(int i),
     * <br><b>i</b>:non-negative
     * <br><b>output</b>:string representation of the i in base
     * 16, starting without a sign.
     */
    public Status Integer0001() {
        boolean failed = false;
          int point[] = {Integer.MIN_VALUE, Integer.MIN_VALUE + 1,
                         -1025, -1024, -1023, -256, -255, -254,
                         -1, 0, 1, 254, 255, 256,
                         1023, 1024, 1025,
                         Integer.MAX_VALUE - 1, Integer.MAX_VALUE};

        for (int i = -100; i < 100; i++) {
          if (!(Integer.toHexString(i).equals(myToHexString(i)))) {
                failed = true;
                ref.println("Failed for :" + i + ": " +
                            myToHexString(i));
            }
        }
        for (int i = 0; i < point.length; i++) {
         if                 (!(Integer.toHexString(point[i]).equals
                           (myToHexString(point[i])))) {
             failed = true;
                ref.println("Failed for :" + point[i]+ ": " +
                          myToHexString(point[i]));
            }
        }
        if (failed) {
            return Status.failed("public static String
                                  toHexString(int i)");
        } else {
            return Status.passed("OKAY");
        }
    }
}
```

Exercise Answer: `ToHexStringTests.html` Test Description

Test Specifications and Descriptions for Integer.toHexString(int)

---

public static String toHexString(int i)

---

public static String toHexString(int i)

Description

Domain testing of input and output conditions, and external pre-conditions for class Integer, method public static String toHexString(int i).

Equivalence Class Partitioning

i	Expected output value	Test Case ID
negative	string representation of the i in base 16, starting with the minus sign	Integer0001
non-negative	string representation of the i in base 16, starting without a sign	Integer0001

Test Descriptions

Test cases included:
Integer0001.

title	Tests for public static String toHexString(int i)
source	ToHexStringTests.java
executeClass	javasoft.sqe.tests.api.java.lang.Integer.ToHexStringTest
keywords	runtime positive

Exercise Answer: GetSuperclassTests.java

CODE EXAMPLE 7 Exercise Answer: GetSuperclassTests.java

```
/*
 * Copyright (c) 1996-2000 Sun Microsystems, Inc.
 * All Rights Reserved.
 *
 * Class getSuperclass Tests
 */

package javasoft.sqe.tests.api.java.lang.Class;

import java.io.PrintWriter;
import com.sun.javatest.Status;
import com.sun.javatest.lib.MultiTest;


public class GetSuperclassTests extends MultiTest {

    /* standalone interface */
    public static void main(String argv[]) {
        GetSuperclassTests test = new GetSuperclassTests();
        test.run(argv, System.err, System.out).exit();
    }

    /**
     * Equivalence class partitioning
     * with state values orientation
     * for public Class getSuperclass(),
     * <br><b>pre-conditions</b>: java.lang.Object class,
     * <br><b>output</b>: null.
     */
    public Status Class0001() {
          try {
           Class o1 = Class.forName("java.lang.Object");
           Class s = o1.getSuperclass();
           if (s == null)
             return Status.passed("OKAY" );
           else
             return Status.failed("unexpected superclass for
                                   Object");
         } catch(ClassNotFoundException e) {
           return Status.failed("forName: ClassNotFoundException
                                thrown");
         }
    }
```

```
/**
 * Equivalence class partitioning
 * with state values orientation
 * for public Class getSuperclass(),
 * <br><b>pre-conditions</b>: any java.lang.Object subclass,
 * <br><b>output</b>: java.lang.Object.
 */
public Status Class0002() {
      try {
       Class o1 =
                 Class.forName("javasoft.sqe.tests.api.java.
                              lang.Class.Test0402");
       Class s = o1.getSuperclass();
       if ((s != null) &&
           (s.getName().equals("java.lang.Object")))
         return Status.passed("OKAY" );
       else
         return Status.failed("no or invalid superclass");
     } catch(ClassNotFoundException e) {
       return Status.failed("forName:
                    ClassNotFoundException thrown");
     }
}

/**
 * Equivalence class partitioning
 * with state values orientation
 * for public Class getSuperclass(),
 * <br><b>pre-conditions</b>: root interface,
 * <br><b>output</b>: null.
 */
public Status Class0003() {
      try {
       Class o1 =
                Class.forName("javasoft.sqe.tests.api.java.
                             lang.Class.Test0403");
       Class s = o1.getSuperclass();
       if (s == null)
         return Status.passed("OKAY" );
       else {

         return Status.failed("unexpected superclass for root
                                       interface");
       }
     } catch(ClassNotFoundException e) {
       return Status.failed("forName:
                   ClassNotFoundException thrown");
     }
}

/**
```

```
 * Equivalence class partitioning
 * with state values orientation
 * for public Class getSuperclass(),
 * <br><b>pre-conditions</b>: non-root interface,
 * <br><b>output</b>: null.
 */
public Status Class0004() {
      try {
         Class o1 =
                  Class.forName("javasoft.sqe.tests.api.
                                java.lang.Class.Test0404");
       Class s = o1.getSuperclass();
       if (s == null)
         return Status.passed("OKAY" );
       else
         return Status.failed("unexpected superclass for
                               non-root interface");
     } catch(ClassNotFoundException e) {
       return Status.failed("forName:
                    ClassNotFoundException thrown");
     }
}

/**
 * Equivalence class partitioning
 * with state values orientation
 * for public Class getSuperclass(),
 * <br><b>pre-conditions</b>: primitive type,
 * <br><b>output</b>: null.
 */
public Status Class0005() {
      Class[] types = {Integer.TYPE, Byte.TYPE, Character.TYPE,
                       Short.TYPE, Integer.TYPE, Long.TYPE,
                       Float.TYPE, Double.TYPE};
       for (int i=0;i<types.length;i++) {
           if (types[i].getSuperclass() != null)
               return Status.failed("unexpected superclass
                                for primitive type");
       }
       return Status.passed("OKAY" );
}

/**
 * Equivalence class partitioning
 * with state values orientation
 * for public Class getSuperclass(),
 * <br><b>pre-conditions</b>: void type,
 * <br><b>output</b>: null.
 */
public Status Class0006() {
       if (Void.TYPE.getSuperclass() != null)
                return Status.failed("unexpected superclass
```

```
                                        for void type");
            return Status.passed("OKAY" );
    }

    /**
     * Equivalence class partitioning
     * with state values orientation
     * for public Class getSuperclass(),
     * <br><b>pre-conditions</b>: any array,
     * <br><b>output</b>: java.lang.Object.
     */
    public Status Class0007() {
            Object[][] a = new Object[1][30];
            Class s = a.getClass().getSuperclass();
            if ((s != null) &&
                (s.getName().equals("java.lang.Object")))
              return Status.passed("OKAY" );
            else
              return Status.failed("no or invalid superclass");
    }

  /**
    * Assertion testing
    * for public Class getSuperclass(),
     * <br><b>pre-conditions</b>: non-immediate java.lang.Object
     * subclass,
     * <br><b>output</b>: correct superclass.
     */
    public Status Class2001() {
          try {
            Class o1 =
                          Class.forName("javasoft.sqe.tests.api.
                              java.lang.Class.Test1402");
            Class s = o1.getSuperclass();
            if ((s != null) &&
                    (s.getName().equals("javasoft.sqe.tests.api.
                            java.lang.Class.Test1402a")))
              return Status.passed("OKAY" );
            else
              return Status.failed("no or invalid superclass");
          } catch(ClassNotFoundException e) {
              return Status.failed("forName:
                           ClassNotFoundException thrown");
          }
    }
}

class Test0402 { int i; }

class Test1402b {
 int b;
}
```

```
class Test1402a extends Test1402b {
 int a;
}

class Test1402 extends Test1402a {
 int ab;
}

interface Test0403 {
 void doIt();
}

interface Test0404 extends Test0403 {
 void doItMore();
}
```

Exercise Answer: `getSuperclass.html` Test Description

Test Specifications and Descriptions for Class.getSuperclass()

---

public Class getSuperclass()

---

public Class getSuperclass()

Description

Domain testing of input and output conditions, and external pre-conditions for class Class, method public Class getSuperclass().

Equivalence Class Partitioning

Pre-conditions	Expected output value	Test Case ID
java.lang.Object class	null	Class0001
any java.lang.Object subclass	java.lang.Object	Class0002
root interface	null	Class0003
non-root interface	null	Class0004
primitive type	null	Class0005
void type	null	Class0006
any array	java.lang.Object	Class0007

Assertion testing

Pre-conditions	Expected output value	Test Case
non-immediate java.lang.Object subclass	output: correct superclass	Class0001

Test Descriptions

Test cases included:

Class0001, Class0002, Class0003, Class0004, Class0005, Class0006, Class0007, Class2001.

title	Class getSuperclass Tests
source	GetSuperclassTests.java
executeClass	javasoft.sqe.tests.api.java.lang.Class.GetSuperclassTests
keywords	runtime positive

APPENDIX **B**

HTML Test Description Code Listings

This appendix contains the HTML code listings for the following files:

`toString.html` Test Description Code

CODE EXAMPLE 8 `toString.html` Test Description Code

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Test Specifications and Descriptions for
   Integer.toString(int, int)</TITLE>
</HEAD>

<BODY>
<H1>Test Specifications and Descriptions for Integer.toString(int,
   int)</H1>
<P>
<HR>
<UL>
  <LI><A HREF="#toString(int,int)"><BIG><CODE>public static String
   toString(int i, int radix)</CODE></BIG></A>
</UL>

<P>
<HR>
```

```
<H3><A NAME="toString(int,int)"><CODE>public static String
   toString(int i, int radix)</CODE></A></H3>

<H4>Description</H4>
<P>
Domain testing of input and output conditions, and external
pre-conditions for class Integer,
method <CODE>public static String toString(int i, int radix)</CODE>.

<H4>Equivalence Class Partitioning</H4>
<TABLE BORDER=1>
  <TR>
    <TH> i
    <TH> radix
    <TH> Expected output value
    <TH> Test Case ID
  <TR>
    <TD> negative
    <TD> less than Character.MIN_RADIX or greater than
   Character.MAX_RADIX
    <TD> string representation of the i in base 16, starting with
   the minus sign
    <TD> testToString
  <TR>
    <TD> negative
    <TD> between Character.MIN_RADIX and Character.MAX_RADIX
    <TD> string representation of the i in base 'radix', starting
   with the minus sign
    <TD> testToString
  <TR>
    <TD> non-negative
    <TD> less than Character.MIN_RADIX or greater than
   Character.MAX_RADIX
   <TD> string representation of the i in base 16, starting without
   a sign
    <TD> testToString
  <TR>
    <TD> non-negative
    <TD> between Character.MIN_RADIX and Character.MAX_RADIX
    <TD> string representation of the i in base 'radix', starting
   without a sign
    <TD> testToString
</TABLE>

<H4>Boundary Value Analysis</H4>
<TABLE BORDER=1>
  <TR>
    <TH> i
    <TH> radix
    <TH> Expected output value
    <TH> Test Case ID
  <TR>
```

```
    <TD> Integer.MIN_VALUE, -1
    <TD> Integer.MIN_VALUE, Character.MIN_RADIX - 1,
   Character.MAX_RADIX + 1, Integer.MAX_VALUE
    <TD> string representation of the i in base 16, starting with
   the minus sign
    <TD> testToString
  <TR>
    <TD> Integer.MIN_VALUE, -1
    <TD> Character.MIN_RADIX, Character.MAX_RADIX
    <TD> string representation of the i in base 'radix', starting
   with the minus sign
    <TD> testToString
  <TR>
    <TD> 0, 1, Integer.MAX_VALUE
    <TD> Integer.MIN_VALUE, Character.MIN_RADIX - 1,
   Character.MAX_RADIX + 1, Integer.MAX_VALUE
   <TD> string representation of the i in base 16, starting without
   a sign
    <TD> testToString
  <TR>
    <TD> 0, 1, Integer.MAX_VALUE
    <TD> Character.MIN_RADIX, Character.MAX_RADIX
    <TD> string representation of the i in base 'radix', starting
   without a sign
    <TD> testToString
</TABLE>

<H4>Test Descriptions</H4>
<A NAME="ToString"></A>
<P>
Test cases included:<br>
  testToString.
<P>
<TABLE BORDER=1 CLASS=TestDescription>
  <TR>
    <TD> <B>title</B>
    <TD> Tests for public static String toHexString(int i)
  <TR>
    <TD> <B>source</B>
    <TD> <A HREF="ToStringTests.java">ToStringTests.java</A>
  <TR>
    <TD> <B>executeClass</B>
    <TD> javasoft.sqe.tests.api.java.lang.Integer.ToStringTests
  <TR>
    <TD> <B>keywords</B>
    <TD> runtime positive
</TABLE>

<P>
<HR>
<EM>&copy; 2001 Sun Microsystems, Inc.  All Rights Reserved.</EM>
</BODY>
```

```
</HTML>
```

getModifiers.html Test Description Code

CODE EXAMPLE 9 getModifiers.html Test Description Code

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Test Specifications and Descriptions for
   Class.getModifiers()</TITLE>
</HEAD>

<BODY>
<H1>Test Specifications and Descriptions for Class.getModifiers()</
   H1>
<HR>
<UL>
  <LI><A HREF="#getModifiers()"><BIG><CODE>public int
   getModifiers()</CODE></BIG></A>
</UL>

<P>
<HR>
<H3><A NAME="getModifiers()"><CODE>public int getModifiers()</
   CODE></A></H3>

<H4>Description</H4>
<P>
Domain testing of input and output conditions, and external
pre-conditions for class Class,
method <CODE>public int getModifiers()</CODE>.

<H4>Equivalence Class Partitioning</H4>
<TABLE BORDER=1>
  <TR>
    <TH> Pre-conditions
    <TH> Expected output value
    <TH> Test Case ID
  <TR>
    <TD> this object: public
    <TD> modifiers with 'public' bit set
    <TD> testCase01
  <TR>
    <TD> this object: non-static
    <TD> modifiers with 'static' bit unset
```

```
    <TD> testCase01
  <TR>
    <TD> this object: final
    <TD> modifiers with 'final' bit set
    <TD> testCase01
  <TR>
    <TD> this object: non-abstract
    <TD> modifiers with 'abstract' bit unset
    <TD> testCase01
  <TR>
    <TD> this object: a class
    <TD> modifiers with 'interface' bit unset
    <TD> testCase01
  <TR>
    <TD> this object: protected
    <TD> modifiers with 'protected' bit set
    <TD> testCase02
  <TR>
    <TD> this object: static
    <TD> modifiers with 'static' bit set
    <TD> testCase02
  <TR>
    <TD> this object: non-final
    <TD> modifiers with 'final' bit unset
    <TD> testCase02
  <TR>
    <TD> this object: private
    <TD> modifiers with 'private' bit unset
    <TD> testCase03
  <TR>
    <TD> this object: package visible
   <TD> modifiers with neither of 'public', 'private' or 'protected'
    bits set
    <TD> testCase04
  <TR>
    <TD> this object: abstract
    <TD> modifiers with 'abstract' bit set
    <TD> testCase04
  <TR>
    <TD> this object: interface
    <TD> modifiers with 'interface' bit set
    <TD> testCase05
</TABLE>

<H4>Assertion testing</H4>
<TABLE BORDER=1>
  <TR>
    <TH> Pre-conditions
    <TH> Assertion
    <TH> Test Case ID
  <TR>
    <TD> this object: array of public classes
```

```
    <TD> If the underlying class is an array class, then its public,
private and protected modifiers are the same as those of
its component type.
    <TD> testCase06
  <TR>
    <TD> this object: array of public classes
    <TD> If this object represents an array class, a primitive type
or void, then its final modifier is always true and its
interface modifer is always false.
    <TD> testCase06
  <TR>
    <TD> this object: array of protected classes
    <TD> If the underlying class is an array class, then its public,
private and protected modifiers are the same as those of
its component type.
    <TD> testCase07
  <TR>
    <TD> this object: array of protected classes
    <TD> If this object represents an array class, a primitive type
or void, then its final modifier is always true and its
interface modifer is always false.
    <TD> testCase07
  <TR>
    <TD> this object: array of private classes
    <TD> If the underlying class is an array class, then its public,
private and protected modifiers are the same as those of
its component type.
    <TD> testCase08
  <TR>
    <TD> this object: array of private classes
    <TD> If this object represents an array class, a primitive type
or void, then its final modifier is always true and its
interface modifer is always false.
    <TD> testCase08
  <TR>
    <TD> this object: array of package-visible classes
    <TD> If the underlying class is an array class, then its public,
private and protected modifiers are the same as those of
its component type.
    <TD> testCase09
  <TR>
    <TD> this object: array of package-visible classes
    <TD> If this object represents an array class, a primitive type
or void, then its final modifier is always true and its
interface modifer is always false.
    <TD> testCase09
  <TR>
    <TD> this object: primitive type
   <TD> If this Class represents a primitive type or void, its public
modifier is always true, and its protected and private modifers
are always false.
    <TD> testCase10
```

```
  <TR>
    <TD> this object: primitive type
    <TD> If this object represents an array class, a primitive type
or void, then its final modifier is always true and its
interface modifer is always false.
    <TD> testCase10
  <TR>
    <TD> this object: void
   <TD> If this Class represents a primitive type or void, its public
modifier is always true, and its protected and private modifers
are always false.
    <TD> testCase11
  <TR>
    <TD> this object: void
    <TD> If this object represents an array class, a primitive type
or void, then its final modifier is always true and its
interface modifer is always false.
    <TD> testCase11
</TABLE>

<H4>Test Descriptions</H4>
<A NAME="GetModifiers"></A>
<P>
Test cases included:<br>
  testCase01,
  testCase02,
  testCase03,
  testCase04,
  testCase05,
  testCase06,
  testCase07,
  testCase08,
  testCase09,
  testCase10,
  testCase11.
<P>
<TABLE BORDER=1 CLASS=TestDescription>
  <TR>
    <TD> <B>title</B>
    <TD> GetModifiers
  <TR>
    <TD> <B>source</B>
   <TD> <A HREF="GetModifiersTests.java">GetModifiersTests.java</A>
  <TR>
    <TD> <B>executeClass</B>
    <TD> javasoft.sqe.tests.api.java.lang.Class.GetModifiersTests
  <TR>
    <TD> <B>keywords</B>
    <TD> runtime positive
</TABLE>
<P>
<HR>
```

```
<EM>
&copy; 2001 Sun Microsystems, Inc.  All Rights Reserved.</EM>
</BODY>
</HTML>
```

ToHexStringTests.html Test Description Code

CODE EXAMPLE 10 ToHexStringTests.html Test Description Code

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Test Specifications and Descriptions for
   Integer.toHexString(int)</TITLE>
</HEAD>

<BODY>
<H1>Test Specifications and Descriptions for
   Integer.toHexString(int)</H1>
<P>
<HR>
<UL>
  <LI><A HREF="#toHexString(int)"><BIG><CODE>public static String
   toHexString(int i)</CODE></BIG></A>
</UL>
<P>
<HR>
<H3><A NAME="toHexString(int)"><CODE>public static String
   toHexString(int i)</CODE></A></H3>

<H4>Description</H4>
<P>
Domain testing of input and output conditions, and external
pre-conditions for class Integer,
method <CODE>public static String toHexString(int i)</CODE>.

<H4>Equivalence Class Partitioning</H4>
<TABLE BORDER=1>
  <TR>
    <TH> i
    <TH> Expected output value
    <TH> Test Case ID
  <TR>
    <TD> negative
    <TD> string representation of the i in base 16, starting with
   the minus sign
```

```
    <TD> Integer0001
  <TR>
    <TD> non-negative
   <TD> string representation of the i in base 16, starting without
   a sign
    <TD> Integer0001
</TABLE>

<H4>Test Descriptions</H4>
<A NAME="ToHexString"></A>
<P>
Test cases included:<br>
  Integer0001.
<P>
<TABLE BORDER=1 CLASS=TestDescription>
  <TR>
    <TD> <B>title</B>
    <TD> Tests for public static String toHexString(int i)
  <TR>
    <TD> <B>source</B>
    <TD> <A HREF="ToHexStringTests.java">ToHexStringTests.java</A>
  <TR>
    <TD> <B>executeClass</B>
    <TD> javasoft.sqe.tests.api.java.lang.Integer.ToHexStringTests
  <TR>
    <TD> <B>keywords</B>
    <TD> runtime positive
</TABLE>

<P>
<HR>
<EM>&copy; 2001 Sun Microsystems, Inc.  All Rights Reserved.</EM>
</BODY>
</HTML>
```

`getSuperclass.html` Test Description Code

CODE EXAMPLE 11 `getSuperclass.html` Test Description Code

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Test Specifications and Descriptions for
   Class.getSuperclass()</TITLE>
</HEAD>
```

```
<BODY>
<H1>Test Specifications and Descriptions for
   Class.getSuperclass()</H1>
<HR>
<UL>
  <LI><A HREF="#getSuperclass()"><BIG><CODE>public Class
   getSuperclass()</CODE></BIG></A>
</UL>

<P>
<HR>
<H3><A NAME="getSuperclass()"><CODE>public Class getSuperclass()</
   CODE></A></H3>

<H4>Description</H4>
<P>
Domain testing of input and output conditions, and external
pre-conditions for class Class,
method <CODE>public Class getSuperclass()</CODE>.

<H4>Equivalence Class Partitioning</H4>
<TABLE BORDER=1>
  <TR>
    <TH> Pre-conditions
    <TH> Expected output value
    <TH> Test Case ID
  <TR>
    <TD> java.lang.Object class
    <TD> null
    <TD> Class0001
  <TR>
    <TD> any java.lang.Object subclass
    <TD> java.lang.Object
    <TD> Class0002
  <TR>
    <TD> root interface
    <TD> null
    <TD> Class0003
  <TR>
    <TD> non-root interface
    <TD> null
    <TD> Class0004
  <TR>
    <TD> primitive type
    <TD> null
    <TD> Class0005
  <TR>
    <TD> void type
    <TD> null
    <TD> Class0006
  <TR>
    <TD> any array
```

```
    <TD> java.lang.Object
    <TD> Class0007
</TABLE>

<H4>Assertion testing</H4>
<TABLE BORDER=1>
  <TR>
    <TH> Pre-conditions
    <TH> Assertion
    <TH> Test Case ID
  <TR>
    <TD> non-immediate java.lang.Object subclass
    <TD> output: correct superclass
    <TD> Class2001
</TABLE>

<H4>Test Descriptions</H4>
<A NAME="GetSuperclass"></A>
<P>
Test cases included:<br>
  Class0001,
  Class0002,
  Class0003,
  Class0004,
  Class0005,
  Class0006,
  Class0007,
  Class2001.
<P>
<TABLE BORDER=1 CLASS=TestDescription>
  <TR>
    <TD> <B>title</B>
    <TD> Class getSuperclass Tests
  <TR>
    <TD> <B>source</B>
   <TD> <A HREF="GetSuperclassTests.java">GetSuperclassTests.java</
    A>
  <TR>
    <TD> <B>executeClass</B>
    <TD> javasoft.sqe.tests.api.java.lang.Class.GetSuperclassTests
  <TR>
    <TD> <B>keywords</B>
    <TD> runtime positive
</TABLE>

<P>
<HR>
<EM>&copy; 2001 Sun Microsystems, Inc.  All Rights Reserved.</EM>
</BODY>
</HTML>
```

APPENDIX C

Introduction to Java Technology API Specifications

The responsibility for writing a Java technology API specification is primarily the concern of the specification writer. However, the compatibility test developer must be able to analyze the specification and provide feedback to the writer in the event of uncovering any flaws that would prevent valid testing.

In order to properly analyze a Java technology API specification for test development purposes, it is helpful to have a familiarity with the structural levels of its components. This appendix briefly introduces the structural components of an API specification.

More in depth information on how to write an API specification can be found at this URL:

http://java.sun.com/j2se/javadoc/writingapispecs/index.html

Specification Users

When participating in the specification development process, it is helpful to remember that an API specification must address the needs of these three distinct types of users:

- Implementors of a given Java API specification
- Compatibility test developers
- Application developers

A complete and accurate specification must adequately address the needs of these three user groups.

Components of an API Specification

An API specification consists of the following types of components, with each specifying a different level of information.

- Top-level specification
- Package specification
- Class and interface specification
- Field specification
- Method specification
- References to external specifications

The remaining sections describe these components.

Top-Level Specification

A top-level specification is composed of those specifications that apply to the entire set of packages. It can include assumptions that underlie the other specifications such as: all objects are presumed to be thread-safe unless otherwise specified.

In addition to the class specific requirements, there are overall Java technology API documentation requirements with respect to handling unchecked exceptions (these are derived from `java.lang.RuntimeException`). It is helpful for the specification writer to include some statements that describe the general situations when a Java application should be prepared to encounter one or more runtime exceptions.

Package Specification

A package specification includes any specifications that apply to the package as a whole or to groups of classes in the package. It must include the following sections:

Executive Summary	A precise and concise description of the package. Useful to describe groupings of classes and introduce major terms.
OS/Hardware Dependencies	Specifies any reliance on the underlying operating system or hardware. For example, the `java.awt` package might describe how the general behavior in that package is allowed to vary from one operating system to another (such as Microsoft Windows, Solaris™ and Macintosh operating systems).

For details about how to actually structure the package information in the `package.html` file according to Sun standards, see this URL:

`http://java.sun.com/products/jdk/javadoc/writingdoccomments/index.html#packagecomments`

Class and Interface Specifications

This section applies to Java classes and interfaces. It may include graphic model diagrams, such as state diagrams to describe static and dynamic information about objects. Code examples are also useful and illustrative.

Each class and interface specification must include the following sections:

Executive Summary	A precise and concise description for the object. Useful to describe groupings of methods and introduce major terms or dependencies.
State Information	Specifies the state information associated with the object, described in a manner that decouples the states from the operations that may query or change these states. This should also include whether instances of this class are thread safe if applicable. (For multi-state objects, a state diagram may be the clearest way to present this information.) If the class allows only single state instances, such as `java.lang.Integer`, this section may be skipped.
OS/Hardware Dependencies	Specifies any reliance on the underlying operating system or hardware.
Allowed Implementation Variances	Specifies how any aspect of this object may vary by implementation. This description should not include information about current implementation bugs.
Security Constraints	If the object has any security constraints or restrictions, an overview of those constraints and restrictions must be provided in the class specification. Documentation for individual security constrained methods must provide detailed information about security constraints.
Serialized Form	This specification ensures that a serialized object can successfully be passed between different implementations of the Java technology. While public classes that implement serializable are part of the serialized form, in some cases it is also necessary to include non-public classes that implement serializable. For more details, see the specific criteria at this URL: `http://java.sun.com/j2se/javadoc/writingapispecs/serialized-criteria.html` The serialized form specification defines the `readObject` and `writeObject` methods, the fields that are serialized, the data types of those fields, and the order those fields are serialized.

Field Specification

Each field specification must include the following sections:

What this field models	Specifies what aspect of the object this field models.
Range of valid values	Specifies all valid and invalid values for this field. For each public and protected static final field whose type is a primitive or String, specify its value. (A future version of the Javadoc software tool will automatically add this value to the specification, but until then the value is manually included in the body of the comment.)
Null value	If this is a reference field, a statement concerning whether this value may be null, and how this object will behave in such a case.

Method Specification

This section applies to Java methods and constructors. Each method and constructor specification must include the following:

Expected Behavior	Specifies the expected or desired behavior of this operation. Describes what aspect of the object being modeled this operation fulfills.
State Transitions	Specifies what state transitions this operation may trigger.
Range of Valid Argument Values	Specifies all valid and invalid values for each argument, including expected behavior for invalid input value or range of values.
Null Argument Values	For each reference type argument, specifies the behavior when null is passed in. NOTE: If possible, document the general null argument behavior at the package or class level, such as causing a `java.lang.NullPointerException` to be thrown. Deviations from this behavior can then be documented at the method level.

Range of Return Values	Specifies the range of possible return values, including where the return value may be null.
Algorithms Defined	When required by the specification, specifies the algorithms used by this operation.
OS/Hardware Dependencies	Specifies any reliance on the underlying operating system or hardware.
Allowed Implementation Variances	Specifies what behavior may vary by implementation. This description should not include information about current implementation bugs.
Cause of Exceptions	Specifies the exceptions thrown by the method, including the argument values, and the state, or context that will cause the specified exception to be thrown. The exceptions thrown from a method need not be mutually exclusive. For more detail about which exceptions should be documented, see this URL: `http://java.sun.com/products/jdk/javadoc/api-specs/throws-tag.html`
Security Constraints	If this operation may be security constrained, this must specify the security check used to constrain this operation. Mention if the method is implemented using an `AccessController.doPrivileged` construct. Must also include a general description of the context or situations where this method may be security constrained.

References to External Specifications

References to external specifications are usually given at either the package level or the class level, as follows.

Package Level References to External Specifications

These are package-wide specifications beyond those generated by Sun or third-parties with the Javadoc application. An example is the UNICODE specification for the `java.text` package. These references can be links to specifications published on the Internet, or titles of specifications available only in print form. The references must be only as narrow or broad in scope as the specification requires. That is, if only a section of a referenced document is considered part of the API specification, then it should link or refer to only that section (refer to the non-specification part of the document as a *related* document). The idea is to clearly delineate what is part of the API specification and what is not.

Class Level References to External Specifications

These are class-level specifications written by Sun or third parties beyond those generated by the Javadoc application. References are not necessary here if they have been included in the package specification.

Java TCK and CTT Glossary

The definitions in this glossary are intended for Java™ Compatibility Test Tools (Java CTT) and Java Technology Compatibility Kits (TCK). Some of these terms may have different definitions or connotations in other contexts. This is a generic glossary covering all of Sun's CTTs and TCKs, and therefore it may contain some terms that are not relevant to the specific product described in this manual.

active agent — A type of *test agent* that initiates a connection to the *JavaTest harness*. Active test agents allow you to run tests in parallel using many agents at once and to specify the test machines at the time you run the tests. Use the *agent monitor* to view the list of registered active agents and synchronize active agents with the JavaTest harness before running tests. See also *test agent*, *passive agent*, and *JavaTest agent*.

active applet instance — An applet instance that is selected on at least one of the logical channels.

agent monitor — The JavaTest window that is used to synchronize *active agents* and to monitor agent activity. The Agent Monitor window displays the agent pool and the agents currently in use.

agents — See *test agent*, *active agent*, *passive agent*, and *JavaTest agent*.

all values — All of the *configuration values* required for a test suite. All values includes the test environment values specific to that test suite and the JavaTest *standard values*.

API member — Fields, methods and constructors for all public classes that are defined in the specification.

API member tests — Tests (sometimes referred to as *class and method* tests) that are designed to verify the semantics of API members.

appeals process — A process for challenging the fairness, validity, accuracy, or relevance of one or more TCK tests. Tests that are successfully challenged are either corrected or added to the TCK's *Exclude List*. See also *first-level appeals process*, *second-level appeals process*, and *Exclude List*.

Application IDentifier (AID) An identifier that is unique in the TCK *namespace*. As defined by ISO 7816-5, it is a string used to uniquely identify card applications and certain types of files in card file systems. An AID consists of two distinct pieces: a 5-byte RID (resource identifier) and a 0 to 11-byte PIX (proprietary identifier extension). The RID is a resource identifier assigned to companies by ISO. The PIX identifiers are assigned by companies. There is a unique AID for each package and a unique AID for each applet in the package. The package AID and the default AID for each applet defined in the package are specified in the CAP file. They are supplied to the converter when the CAP file is generated.

Application Management Software (AMS) Software used to download, store and execute Java applications. Another name for AMS is *Java Application Manager (JAM)*.

Application Programming Interface (API) An API defines calling conventions by which an application program accesses the operating system and other services.

Application Protocol Data Unit (APDU) A script that gets sent to the test applet as defined by ISO 7816-4.

assertion A statement contained in a structured Java technology API specification to specify some necessary aspect of the API. Assertions are statements of required behavior, either positive or negative, that are made within the *Java technology specification*.

assertion testing Compatibility testing based on testing assertions in a specification.

automatic tests Test that run without any intervention by a user. Automatic tests can be queued up and run by the *test harness* and their results recorded without anyone being present.

behavior-based testing A set of test development methodologies that are based on the description, behavior, or requirements of the system under test, not the structure of that system. This is commonly known as “black-box” testing.

boundary value analysis A test case development technique which entails developing additional test cases based on the boundaries defined by previously categorized equivalence classes.

class The prototype for an object in an *object-oriented* language. A class may also be considered a set of objects which share a common structure and behavior. The structure of a class is determined by the class variables which represent the state of an object of that class and the behavior is given by a set of methods associated with the class. See also *classes*.

classes Classes are related in a class hierarchy. One class may be a specialization (a “subclass”) of another (one of its “superclasses”), may be composed of other classes, or may use other classes in a client-server relationship. See also *class*.

compatibility rules Define the criteria a Java technology implementation must meet in order to be certified as "compatible" with the technology specification. See also *compatibility testing*.

compatibility testing The process of testing an implementation to make sure it is compatible with the corresponding Java technology specification. A suite of tests contained in a *Technology Compatibility Kit (TCK)* is typically used to test that the implementation meets and passes all of the *compatibility rules* of that specification.

configuration Information about your computing environment required to execute a *Technology Compatibility Kit (TCK)* test suite. The *JavaTest harness* version 3.x uses a *configuration interview* to collect and store configuration information. The *JavaTest harness* version 2.x uses *environment files* and *parameter files* to obtain configuration data.

configuration editor The dialog box used by *JavaTest harness* version 3.x to present the *configuration interview*.

configuration interview A series of questions displayed by the *JavaTest harness* version 3.x to gather information from the user about the computing environment in which the TCK is being run. This information is used to produce a *test environment* that the JavaTest harness uses to execute tests.

configuration value Information about your computing environment required to execute a TCK test or tests. The *JavaTest harness* version 3.x uses a *configuration interview* to collect configuration values. The JavaTest harness version 2.x uses *environment files* and *parameter files* to obtain configuration data.

domain See *security domain*.

environment files Files used by the *JavaTest harness* 2.x to configure how the JavaTest harness runs the tests on your system. Environment files have the filename extension of `.jte`. Environment files are used in conjunction with *parameter files*.

equivalence class partitioning A test case development technique which entails breaking a large number of test cases into smaller subsets with each subset representing an equivalent category of *test cases*.

Exclude List A list of TCK tests that a *technology implementation* is not required to pass in order to certify compatibility. The *JavaTest harness* uses exclude list files (`*.jtx`), to filter out of a test run those tests that do not have to be passed. The exclude list provides a level playing field for all implementors by ensuring that when a test is determined to be invalid, no implementation is required to pass it. Exclude lists are maintained by the *Maintenance Lead (ML)* and are made available to all technology licensees. The ML may add tests to the exclude list for the test suite as needed at any time. An updated exclude list replaces any previous exclude lists for that test suite.

first-level appeals process — The process by which a technology implementor can appeal or challenge a TCK test. First-level appeals are resolved by the Expert Group responsible for the technology specification and TCK. See also *appeals process* and *second-level appeals process*.

framework — See *test framework*.

Graphical User Interface (GUI) — Provides application control through the use of graphic images.

HTML test description — A *test description* that is embodied in an HTML table in a file separate from the test source file.

implementation — See *technology implementation*.

instantiation — In object-oriented programming, means to produce a particular object from its class template. This involves allocation of a data structure with the types specified by the template, and initialization of instance variables with either default values or those provided by the class's constructor function.

interactive tests — Tests that require some intervention by the user. For example, the user might have to provide some data, perform some operation, or judge whether or not the implementation passed or failed the test.

Java™ 2, Standard Edition (J2SE™) platform — A set of specifications that defines the desktop runtime environment required for the deployment of Java applications. The J2SE implementations are available for a variety of platforms, but most notably the Sun Solaris operating environment and Microsoft Windows.

Java Application Manager (JAM) — A native application used to download, store and execute Java applications. Another name for JAM is *Application Management Software (AMS)*.

Java Archive (JAR) — A platform-independent file format that combines many files into one.

Java Compatibility Test Tools (Java CTT) — Tools, documents, templates, and samples that can be used to design and build TCKs. Using the Java CTT simplifies compatibility test development and makes developing and running tests more efficient.

Java Community Process (JCP) — An open organization of international Java developers and licensees whose charter is to develop and revise *Java technology specifications*, and their associated *Reference Implementation (RI)*, and *Technology Compatibility Kit (TCK)*.

Java Platform Libraries — The class libraries that are defined for each particular version of a *Java technology* in its *Java technology specification*.

Java Specification Request (JSR) The actual descriptions of proposed and final technology specifications for the Java platform.

Java technology A Java technology is defined as a *Java technology specification* and its *Reference Implementation (RI)*. Examples of Java technologies are Java 2 Platform, Standard Edition (J2SE™), the Connected Limited Device Configuration (CLDC), and the Mobile Information Device Profile (MIDP).

Java Technology Compatibility Kit See *Technology Compatibility Kit (TCK)*.

Java technology specification A written specification for some aspect of the *Java technology*.

JavaTest agent A *test agent* supplied with the *JavaTest harness* to run TCK tests on a Java implementation where it is not possible or desirable to run the main JavaTest harness. See also *test agent*, *active agent*, and *passive agent*.

JavaTest harness A *test harness* that has been developed by Sun to manage test execution and result reporting for a *Technology Compatibility Kit (TCK)*. The harness configures, sequences, and runs test suites. The JavaTest harness is designed to provide flexible and customizable test execution. It includes everything a test architect needs to design and implement tests for Java technology specifications.

keywords Used to direct the *JavaTest harness* to include or exclude tests from a test run. Keywords are defined for a *test* by the *test suite* architect.

Maintenance Lead (ML) The person responsible for maintaining an existing *Java technology specification* and related *Reference Implementation (RI)* and *Technology Compatibility Kit (TCK)*. The ML manages the TCK *appeals process*, *Exclude List*, and any revisions needed to the specification, TCK, or RI.

methods Procedures or routines associated with one or more *class*, in *object-oriented* languages.

MultiTest A JavaTest library class that enables tests to include multiple *test cases*. Each test case can be addressed individually in a test suite *Exclude List*.

namespace A set of names in which all names are unique.

object-oriented A category of programming languages and techniques based on the concept of *objects* which are data structures encapsulated with a set of routines, called *methods*, which operate on the data.

objects In *object-oriented* programming, objects are unique instances of a data structure defined according to the template provided by its *class*. Each object has its own values for the variables belonging to its class and can respond to the messages (*methods*) defined by its class.

packages A *namespace* within the Java programming language. It can have *class* and interfaces. A package is the smallest unit within the Java programming language.

parameter files Files used by the *JavaTest harness* 2.x to configure individual test runs. For JavaTest harness version 2.x parameter files have the filename extension `*.jtp`. (JavaTest harness version 3.x does not use parameter files, but does use files with the filename extension `*.jti` to store test harness configurations.)

passive agent A type of *test agent* that must wait for a request from the *JavaTest harness* before they can run tests. The JavaTest harness initiates connections to passive agents as needed. See also *test agent*, *active agent*, and *JavaTest agent*.

prior status A JavaTest filter used to restrict the set of tests in a test run based on the last test result information stored in the test result files (`.jtr`).

Profile Specification A specification that references one of the Platform Edition Specifications and zero or more other Java technology specifications (that are not already a part of a Platform Edition Specification). APIs from the referenced Platform Edition must be included according to the referencing rules set out in that Platform Edition Specification. Other referenced specifications must be referenced in their entirety.

Program Management Office (PMO) The administrative structure that implements the *Java Community Process (JCP)*.

protected API Some APIs, called protected APIs, require that an applet have permission to access them. An attempt to use a protected API without the necessary permissions cause a security exception error.

protection domain A set of permissions that control which protected APIs an applet can use.

Reference Implementation (RI) The prototype or proof of concept implementation of a *Java technology specification*. All new or revised specifications must include an RI. A specification RI must pass all of the TCK tests for that specification.

second-level appeals process Allows technology implementors who are not satisfied with a first-level appeal decision to appeal the decision. See also *appeals process* and *first-level appeals process*.

security domain A set of permissions that define what an application is allowed to do in relationship to restricted APIs and secure communications.

security policy The set of permissions that a *technology implementation* or *Application Programming Interface (API)* requires an application to have in order for the application to access the implementation or API.

signature file A text representation of the set of public and protected features provided by an API that is part of a finished TCK. It is used as a signature reference during the TCK signature test for comparison to the technology implementation under test.

signature test Checks that all the necessary API members are present and that there are no extra members which illegally extend the API. It compares the API being tested with a reference API and confirms if the API being tested and the reference API are mutually binary compatible.

specification See *Java technology specification*.

standard values A *configuration value* used by the JavaTest harness to determine which tests in the test suite to run and how to run them. The user can change standard values using either the *all values* or Standard Values view in the *configuration editor*.

structure-based testing A set of test development methodologies that are based on the internal structure or logic of the system under test, not the description, behavior, or requirements of that system. This is commonly known as "white-box" or "glass-box" testing. Compatibility testing does not make use of structure-based test techniques.

system configuration Refers to the combination of operating system platform, Java programming language, and JavaTest harness tools and settings.

tag test description A *test description* that is embedded in the Java language source file of each test.

Technology Compatibility Kit (TCK) The suite of tests, tools, and documentation that allows an implementor of a *Java technology specification* to determine if the implementation is compliant with the specification.

TCK coverage file Used in most TCKs developed by Sun to track the test coverage of a *test suite* during test development. They bind test cases to their related *assertion* in the *technology specification* that is being implemented and tested. The bindings make it possible to generate statistical reports on test coverage.

technology implementation Any binary representation of the form and function defined by a *Java technology specification*.

technology specification See *Java technology specification*.

test agent A Java application that receives tests from the *test harness*, runs them on the implementation being tested, and reports the results back to the test harness. Test agents are normally only used when the TCK and implementation being tested are running on different platforms. See also *active agent*, *passive agent*, and *JavaTest agent*.

test The source code and any accompanying information that exercise a particular feature, or part of a feature, of a *technology implementation* to make sure that the feature complies with the *Java technology specification*'s compatibility rules.

A single test may contain multiple *test cases*. Accompanying information may include test documentation, auxiliary data files, or other resources used by the source code. Tests correspond to assertions of the specification.

test cases A small test that is run as part of a set of similar *test*s. Test cases are implemented using the JavaTest *MultiTest* library class. A test case tests a specification assertion, or a particular feature, or part of a feature, of an assertion.

test command A class that knows how to execute test classes in different environments. Test commands are used by the *test script* to execute tests.

test command template A generalized specification of a *test command* in a *test environment*. The test command is specified in the test environment using variables so that it can execute any test in the test suite regardless of its arguments.

test description Machine readable information that describes a test to the *test harness* so that it can correctly process and run the related test. The actual form and type of test description depends on the attributes of the *test suite*. A test description exists for every test in the test suite and is read by the *test finder*. When using the *JavaTest harness*, the test description is a set of test-suite-specific name/value pairs in either HTML tables or Javadoc™-style tags.

test environment Consists of one or more *test command template* that the *test script* uses to execute tests and set of name/value pairs that define *test description* entries or other values required to run the tests.

test execution model The steps involved in executing the tests in a *test suite*. The test execution model is implemented by the *test script*.

test finder When using the *JavaTest harness*, a nominated class, or set of classes, that read, verify, and process the files that contain *test description* in a *test suite*. All test descriptions that are located or found are handed off to the JavaTest harness for further processing.

test framework Software designed and implemented to customize a test harness for a particular test environment. In many cases test framework components have to be provided by the TCK user. In addition to the *test harness*, a test framework might (or might not) include items such as a: *configuration interview*, *Java Application Manager (JAM)*, *test agent*, *test finder*, *test script*, and so forth. A test framework might also include other user-supplied software components (plug-ins) to provide support for implementation-specific protocols.

test harness The applications and tools that are used for test execution and *test suite* management. The *JavaTest harness* is an example of a test harness.

test script A Java class whose job it is to interpret the *test description* values, run the tests, and then report the results back to the *JavaTest harness*. The test script must understand how to interpret the test description information returned to it by the *test finder*.

test specification A human-readable description, in logical terms, of what a test does and the expected results. Test descriptions are written for test users who need to know in specific detail what a test does. The common practice is to write the test specification in HTML format and store it in the *test suite*'s test directory tree.

test suite A collection of tests, used in conjunction with the *test harness* to verify compliance of the licensee's implementation of a *Java technology specification*. Every *Technology Compatibility Kit (TCK)* contains one or more test suites.

work directory A directory associated with a specific test suite and used by the *JavaTest harness* to store files containing information about the test suite and its tests.

Index

Symbols

A

B

C

D

E

Q

R

S

T

W