Java Compatibility Kit 6b User's Guide

For Java Technology Licensees

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.1-800-555-9SUN or 1-650-960-1300

August 2008

080822@20795

Contents

Figures

Tables

Examples

Preface

The Java™ Compatibility Kit (JCK) is a portable, configurable automated test suite. As such, it requires preparation and study prior to execution.

Java technology licensees can create Java Platform, Standard Edition 6 (Java SE 6) software components for their own platforms by following the Java Community Process℠ (JCP℠) program specifications. To ensure that your products comply with the Java specifications, Sun developed the JCK.

This manual describes how to use JCK 6b to test your implementation of the Java platform. It details the certification procedure, the components of the test suite, and the specifics of configuring the JavaTest™ harness for specific portions of the test suite.

In most cases, you cannot unpack JCK 6b and immediately run tests. Depending on your test environment, you might be required to configure a system to be a server for the JavaTest harness or to configure one or more systems for networking tests. Before running the JCK tests, you must understand how to use the JavaTest harness to configure and run tests. JCK 6b provides a tutorial and sample tests suites that you can use to become familiar with using the JavaTest harness to configure and run tests on an existing Java platform reference implementation.

See Appendix D for detailed information about running the tutorial and sample tests. After you are familiar with using the JavaTest harness to configure and run tests, you can use JCK 6b to test your target system.

Before You Read This Guide

This guide assumes that you are familiar with the Java programming language and with running Java technology applications on at least one platform implementation. You must also understand how to configure the JavaTest harness and the computer system used in testing. See "Related Documentation" on page 23 for a list of the JavaTest harness user documents.

How This Guide Is Organized

- Chapter 1 introduces JCK 6b, explains the purpose of conformance testing, and describes how to obtain JCK 6b.
- Chapter 2 describes the procedures and requirements for certifying your software.
- Chapter 3 describes how to install JCK 6b and provides JCK 6b release information.
- Chapter 4 describes how to start running JCK 6b tests using the JavaTest harness.
- Chapter 5 contains setup and runtime information about specific tests.
- Chapter 6 provides information and guidance about debugging test problems.
- Chapter 7 describes how to run the JavaTest agent with limited resources.
- Appendix A contains the JCK test specification that provides detailed information about the test description and the execution models.
- Appendix B contains platform-specific examples.
- Appendix C contains an example of setting up a Kerberos Key Distribution Center.
- Appendix D contains a tutorial that you can use to familiarize yourself with the JavaTest harness before running the JCK 6b test suite.

UNIX® Platform Commands

This document does not contain information about basic UNIX platform commands and procedures such as shutting down the system, booting the system, and configuring devices.

See one or more of the following for this information:

- *Solaris Handbook for Sun Peripherals*
- *AnswerBook2™* software online documentation for the Solaris™ Operating System
- Other software documentation that you received with your system

Typographic Conventions

This User's Guide uses the following typographic conventions:

TABLE P–1 Typographic Conventions

Typeface	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`
AaBbCc123	What you type, when contrasted with onscreen computer output	`%` **su** `Password:`
AaBbCc123	Book titles, new words or terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this.
aabbcc123	Command-line variable; replace with a real name or value.	To delete a file, type `rm` *filename*..

Shell Prompts in Command Examples

Examples in this User's Guide might contain the following shell prompts:

TABLE P–2 Shell Prompts

Shell	Prompt
C shell	`machine_name%`
C shell for superuser	`machine_name#`
Bourne shell and Korn shell	`$`
Bourne shell and Korn shell for superuser	`#`

Related Documentation

When installed, JCK 6b includes a `doc` directory that contains both JCK 6b and JavaTest harness documentation in PDF and HTML format.

TABLE P–3 Related Documentation Titles

Application	Title
JavaTest harness	*JavaTest Harness User's Guide: Graphical User Interface* *JavaTest Harness User's Guide: Command-Line Interface* *JavaTest Harness Agent User's Guide*
Programming Reference	*The Java Programming Language*
Programming Reference	*The Java Language Specification*

Accessing Sun Documentation Online

The Java Developer Connection (`http://developer.java.sun.com/developer/infodocs/`)[SM] web site enables you to access Java platform technical documentation.

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. You can email your comments to Sun at `docs@java.sun.com`.

◆ ◆ ◆ CHAPTER 1

1 Introduction

The JCK 6b is a test suite and a set of tools used to certify that a Java platform implementation conforms to the applicable Java platform specifications and to Sun's reference implementations. The end result is a Java platform implementation certified to bear the Java Compatible™ logo. The test suites exercise assertions described in the specifications and implemented in the JDK™ software. Tests run on any Java platform implementation, from powerful servers to small hand-held devices. JCK 6b uses the JavaTest harness and test suite management tools.

1.1 Conformance Defined

Conformance testing differs from traditional product testing in a number of ways. Following are some of the features and areas of the Java platform on which conformance testing focuses:

- Features that are likely to differ across implementations
- Features that rely on hardware or operating system-specific behavior
- Features that are difficult to port
- Features that either mask or abstract hardware behavior
- Features that either mask or abstract operating system behavior

Conformance test development for a given feature relies on a complete specification and reference implementation for that feature. Conformance testing is not primarily concerned with robustness, performance, or ease of use. In fact, using the JCK 6b test suite in repeated test runs as is typically done in product testing is time consuming for certain parts of the suite.

While JCK 6b is called a “compatibility test suite,” it really tests Java platform implementation conformance of the applicable Java platform specifications with Sun's reference implementation. Conformance testing is a means of ensuring correctness, completeness, and consistency across all Java platform implementations. These three attributes are necessary for the Java platform to be successful.

1.2 Why Conformance Is Important

Java platform conformance is important to different groups involved with Java technologies for different reasons. Conformance testing is used by Sun to ensure that the Java platform does not become fragmented as it is ported to different operating systems and hardware environments (such as Microsoft 2000 and XP (Win32) and Solaris platform). Conformance testing benefits developers working in the Java programming language, allowing them to write applications once and to deploy them across heterogeneous computing environments without porting. Conformance testing allows application users to obtain applications from disparate sources and deploy them with confidence. Conformance testing benefits Java platform implementors by ensuring a level playing field for all Java platform ports.

1.3 Learning Curve

Conformance testing, just as with many other software testing tasks, is an iterative process. You must run JCK 6b many times in the process of certifying a Java platform port. The learning curve for the JCK is somewhat steep because it is a complex test suite. JCK 6b test tools and test suite are complex because they must perform the following functions:

- Work correctly on all platforms that support porting of the Java programming language
- Work correctly in a number of different environments, such as applets applications, and servers without graphical user interfaces (GUIs)
- Work correctly on a number of different types of systems, such as palm-top systems, network computers, desktop systems, workstations, and servers
- Work in all test environments without being ported to any of them

1.4 About the Tests

All tests in JCK 6b are based on the written specifications for the Java platform and tested against Sun's reference implementations. Any test that is not specification based, or is vaguely defined by the specification, can be appealed and might be excluded. Any test that is believed to be implementation dependent (based on a particular thread scheduling model, a particular file system behavior, and so on) can be appealed and might be excluded. See Chapter 2 for a complete discussion of the test appeals process and the exclude list.

Most tests in JCK 6b are self checking. Some tests require tester interaction. Each test returns either a Pass or Fail status. For a given platform to be certified, all of the required tests must have passing results. The definition of required tests change over time and might change from platform to platform. Before your final certification test passes, you must download and use the latest exclude list for JCK 6b.

1.5 Obtaining a Java Compatibility Kit

JCK 6b is available by download from the Java technology licensee web site. The most current exclude lists are also available by download from this site. Contact your Java Licensee Engineering representatives for any other information.

◆ ◆ ◆ CHAPTER 2

Procedure for Java SE 6 Technology Certification

This chapter describes the Java SE 6 technology compatibility testing procedure and compatibility requirements for JCK 6b.

This chapter contains the following sections:

2.1 Certification Overview

The certification process for the Java SE 6 technology consists of the following activities:

- Install JCK 6b and execute it in accordance with the instructions in this User's Guide.
- Ensure that you meet the requirements outlined in "2.2 Compatibility Requirements" on page 30.
- Certify to Java Licensee Engineering that you have finished testing and that you meet all the compatibility requirements.

2.2 Compatibility Requirements

The compatibility requirements for the Java SE 6 technology consist of meeting the requirements set forth by the rules and associated definitions contained in this section.

2.2.1 Definitions

These definitions are for use only with these compatibility requirements and are not intended for any other purpose.

TABLE 2–1 Definitions

Term	Definition
Compiler	A software development tool that implements or incorporates a compilation function from source code written in the Java programming language, as specified by Sun, into Executable Output.
Compiler Product	A licensee product in which a Compiler is implemented or incorporated, which is subject to compatibility testing.
Computational Resource	A piece of hardware or software that may vary in quantity, existence, or version, which may be required to exist in a minimum quantity and/or at a specific or minimum revision level so as to satisfy the requirements of the Test Suite. Examples of computational resources that may vary in quantity are RAM and file descriptors. Examples of computational resources that may vary in existence (that is, may or may not exist) are graphics cards and device drivers. Examples of computational resources that may vary in version are operating systems and device drivers.
Conformance Tests	All tests in the Test Suite for an indicated Technology Under Test, as distributed by the Maintenance Lead, excluding those tests on the Exclude List for the Technology Under Test.
Development Kit	A software product that implements or incorporates a Compiler, a Schema Compiler, a Schema Generator, a Java-to-WSDL Tool, a WSDL-to-Java Tool, and an RMI Compiler.
Documented	Made technically accessible and made known to users, typically by means such as marketing materials, product documentation, usage messages, or developer support programs.

TABLE 2–1 Definitions *(Continued)*

Term	Definition
Edition	A Version of the Java Platform. Editions include Java Platform Standard Edition and Java Platform Enterprise Edition.
Endorsed Standard	A Java API defined through a standards process other than the Java Community Process. The Endorsed Standard packages are referenced later in this chapter.
Exclude List	The most current list of tests, distributed by the Maintenance Lead, that are not required to be passed to certify conformance. The Maintenance Lead may add to the Exclude List for that Test Suite as needed at any time. The updated Exclude List supplants any previous Exclude Lists for that Test Suite.
Executable Output	The output of a Compiler or other tools intended to be executed by a Runtime Interpreter. For example, an error message generated by a negative test in the Conformance Tests for a Compiler is not Executable Output.
Java-to-WSDL Output	Output of a Java-to-WSDL Tool that is required for Web service deployment and invocation.
Java-to-WSDL Tool	A software development tool that implements or incorporates a function that generates web service endpoint descriptions in WSDL and XML schema format from Source Code as specified by the JAXWS Specification.
Libraries	The class libraries, as specified through the Java Community Process (JCP), for the Technology Under Test. The Libraries for the Java SE 6 technology are listed at the end of this chapter.
Location Resource	A location of classes or native libraries that are components of the test tools or tests, such that these classes or libraries may be required to exist in a certain location in order to satisfy the requirements of the test suite. For example, classes may be required to exist in directories named in a `CLASSPATH` variable, or native libraries may be required to exist in directories named in a `PATH` variable.
Maintenance Lead	The Java Community Process member responsible for maintaining the Specification, reference implementation, and TCK for the Technology. Sun is the Maintenance Lead for Java SE 6 technology.
Native Code	Code intended to be executed directly on the hardware platform on which a Runtime is executed.

TABLE 2–1 Definitions *(Continued)*

Term	Definition
Operating Mode	Any Documented option of a Product that can be changed by a user in order to modify the behavior of the Product. For example, an Operating Mode of a Runtime can be binary (enable/disable optimization), an enumeration (select from a list of localizations), or a range (set the initial Runtime heap size). Note that an Operating Mode may be selected by a command line switch, an environment variable, a GUI user interface element, a configuration or control file, etc.
Product	A Compiler Product or a Runtime Product.
Product Configuration	A specific setting or instantiation of an Operating Mode. For example, a Runtime supporting an Operating Mode that permits selection of an initial heap size might have a Product Configuration that sets the initial heap size to 1 Mb.
Reference Compiler	A Compiler designated in this document as the reference implementation for the Technology Under Test.
Reference Runtime	A Runtime designated in this document as the reference implementation for the Technology Under Test.
Resource	A Computational Resource, a Location Resource, or a Security Resource.
RMI Compiler	A software development tool that implements or incorporates a function that generates helper classes for RMI protocols from remote object implementation classes.
Rules	These definitions and rules in this Compatibility Requirements section of this User's Guide.
Runtime	The combination of Runtime Interpreter and an implementation of the Libraries for all of the Technologies included in the Product.
Runtime Interpreter	A program that implements the Java Virtual Machine (JVM™) for the Technology. **Note –** The terms "Java Virtual Machine" and "JVM" mean a Virtual Machine for the Java™ platform.
Runtime Product	A licensee product in which a Runtime, and optionally a Development Kit, is implemented or incorporated, which is subject to compatibility testing.

TABLE 2–1 Definitions *(Continued)*

Term	Definition
Schema Compiler	A software development tool that implements or incorporates a compilation function from the schema source code written in the W3C XML Schema language [XSD Part 1][XSD Part 2] into Source Code or compiled classes that represent XML data described by the schema.
Schema Compiler Output	The output of a Schema Compiler intended to be run in combination with the Libraries.
Schema Generator	A software development tool that implements or incorporates a function that maps Source Code or class files to Schema Generator Output.
Schema Generator Output	The output of Schema Generator intended to be a W3C XML Schema defined in the XML Schema recommendation [XSD Part 1][XSD Part 2]. The output of a Schema Generator that is in the format defined by the W3C XML Schema recommendation [XSD Part 1][XSD Part 2].
Security Resource	A security privilege or policy necessary for the proper execution of the Test Suite. For example, the user executing the Test Suite will need the privilege to access the files and network resources necessary for use of the Product.
Source Code	Source code written in the Java programming language.
Specifications	The documents produced through the Java Community Process that define a particular Version of a Technology. The Specifications for the Technology Under Test can be found later in this chapter.
Technology	Specifications and a reference implementation produced through the Java Community Process.
Technology Under Test	Specifications and the reference implementation for the Java SE 6 technology.
Test Suite	The requirements, tests, and testing tools distributed by the Maintenance Lead as applicable to a given Version of the Technology.
Version	A release of the Technology, as produced through the Java Community Process services.
WSDL-to-Java Output	Output of a WSDL-to-Java tool that is required for Web service deployment and invocation.

TABLE 2–1 Definitions *(Continued)*

Term	Definition
WSDL-to-Java Tool	A software development tool that implements or incorporates a function that generates web service interfaces for clients and endpoints from a WSDL description as specified by the JAXWS Specification.

2.2.2 Rules for Java SE 6 Technology Products

The following rules apply for each version of an operating system, software component, and hardware platform Documented as supporting the Product:

1. The Product must be able to satisfy all applicable compatibility requirements, including passing all Conformance Tests, in every Product Configuration and in every combination of Product Configurations, except only as specifically exempted by these Rules.

   For example, if a Product provides distinct Operating Modes to optimize performance, then that Product must satisfy all applicable compatibility requirements for a Product in each Product Configuration, and combination of Product Configurations, of those Operating Modes.

   a. If an Operating Mode controls a Resource necessary for the basic execution of the Test Suite, testing may always use a Product Configuration of that Operating Mode providing that Resource, even if other Product Configurations do not provide that Resource. Notwithstanding such exceptions, each Product must have at least one set of Product Configurations of such Operating Modes that is able to pass all the Conformance Tests.

      For example, a Product with an Operating Mode that controls a security policy (i.e., Security Resource) which has one or more Product Configurations that cause Conformance Tests to fail may be tested using a Product Configuration that allows all Conformance Tests to pass.

   b. A Product Configuration of an Operating Mode that causes the Product to report only version, usage, or diagnostic information is exempted from these compatibility rules.

   c. A Product may contain an Operating Mode that selects the Edition with which it is compatible. The Product must meet the compatibility requirements for the corresponding Edition for all Product Configurations of this Operating Mode. This Operating Mode must affect no smaller unit of execution than an entire application.

   d. A Runtime Product may have Operating Modes that enable the optional Java Virtual Machine Tool Interface (JVM TI) functionality of the Java Platform Debugger Architecture (JPDA) technology. If JVM TI functionality is provided, the Product must pass the conformance tests that test JVM TI. Testing of JVM TI functionality may always use a Product Configuration that enables the functionality.

e. A Runtime Product may have Operating Modes that enable the optional Java Debug Wire Protocol (JDWP) functionality of the JPDA technology. If JDWP functionality is provided, the Product must pass the conformance tests that test JDWP. Testing of JDWP functionality may always use a Product Configuration that enables the functionality.

f. A Runtime Product may have Operating Modes that enable the required java.lang.instrument functionality. Testing of such functionality may always use a Product Configuration that enables the functionality.

2. Some Conformance Tests may have properties that may be changed. Properties that can be changed are identified in the configuration interview. Apart from changing such properties and other allowed modifications described in this User's Guide (if any), no source or binary code for a Conformance Test may be altered in any way without prior written permission. Any such allowed alterations to the Conformance Tests would be posted to the Java Licensee Engineering web site and apply to all licensees.
3. The testing tools supplied as part of the Test Suite or as updated by the Maintenance Lead must be used to certify compliance.
4. The Exclude List associated with the Test Suite cannot be modified.
5. The Maintenance Lead can define exceptions to these Rules. Such exceptions would be made available to and apply to all licensees.
6. All hardware and software component additions, deletions, and modifications to a Documented supporting hardware/software platform, that are not part of the Product but required for the Product to satisfy the compatibility requirements, must be Documented and available to users of the Product.

   For example, if a patch to a particular version of a supporting operating system is required for the Product to pass the Conformance Tests, that patch must be Documented and available to users of the Product.
7. A Runtime Product must contain the full set of public and protected classes and interfaces for all the Libraries. Those classes and interfaces must contain exactly the set of public and protected methods, constructors, and fields defined by the Specifications for those Libraries. No subsetting, supersetting, or modifications of the public and protected API of the Libraries are allowed except only as specifically exempted by these Rules.

   a. If a Product includes Technologies in addition to the Technology Under Test, then it must contain the full set of combined public and protected classes and interfaces. The API of the Product must contain the union of the included Technologies. No further modifications to the APIs of the included Technologies are allowed.

   b. A Product may provide a newer version of an Endorsed Standard. Upon request, the Maintenance Lead will make available alternate Conformance Tests as necessary to conform with such newer version of an Endorsed Standard. Such alternate tests is to be made available to and apply to all licensees. If a Product provides a newer version of an Endorsed Standard, the version of the Endorsed Standard supported by the Product must be Documented.

c. The Maintenance Lead may authorize the use of newer Versions of a Technology included in the Technology Under Test. A Product that provides a newer Version of a Technology must meet the Compatibility Requirements for that newer Version, and must Document that it supports the newer Version.

   For example, the Java SE 6 technology Maintenance Lead authorized use of a newer version of a Java technology, Java API for XML Processing (JAXP) that is also available independently of Java SE 6 technology and is evolving quickly to meet market demands such as supporting the latest XML language features.

8. Except for tests specifically required by this TCK to be rebuilt (if any), the binary and sources of Conformance Tests supplied as part of the Test Suite or as updated by the Maintenance Lead must be used to certify compliance.
9. The functional programmatic behavior of any binary class or interface must be that defined by the Specifications.
10. If a Runtime Product supplies neither 1) a Documented interface that allows a program running in the Runtime to directly interact with Native Code nor 2) a Documented interface that allows Native Code to directly interact with a program running in the Runtime, then the Product is not required to pass the Conformance Tests that test the JNI implementation. If, however, the Product supplies either (1) or (2) above, the Product must pass the Conformance Tests that test the JNI implementation.

2.2.3 Rules for Compiler Products

These Rules, in addition to the Rules for "2.2.2 Rules for Java SE 6 Technology Products" on page 34 Products, apply to all products that include a Compiler.

1. For each Product Configuration that causes the Compiler to generate Executable Output for some Version of the Technology, the Compiler, when run in that Product Configuration, must be able to satisfy all applicable compatibility requirements, including passing all Conformance Tests for a Compiler associated with that Version.

   For example, if a Compiler has a Product Configuration that causes it to generate Executable Output for Version 1.2 of the Technology, then that Compiler, when run in that Product Configuration, must satisfy all applicable compatibility requirements and pass the Conformance Tests for a Version 1.2 Compiler. If that Compiler also has a Product Configuration that causes it to generate Executable Output for Version 1.1 of the Technology, then it must also, when run in this second Product Configuration, satisfy all applicable compatibility requirements and pass the Conformance Tests for a Version 1.1 Compiler.
2. For each Product Configuration that causes the Compiler to generate Executable Output for some Version of the Technology, the Executable Output of the Compiler, when the Compiler is run in that Product Configuration, must execute properly on a Reference Runtime associated with that Version.

For example, if a Compiler has a Product Configuration that causes it to generate Executable Output for Version 1.2 of the Technology, then that Executable Output must execute properly on the Reference Runtime associated with Version 1.2 of the Technology. If that Compiler also has a Product Configuration that causes it to generate Executable Output for Version 1.1 of the Technology, then that Executable Output must execute properly on the Reference Runtime associated with Version 1.1 of the Technology.

3. The Compiler must not produce Executable Output from source code that does not conform to the Specifications.

   For example, a Compiler must not produce Executable Output from source code that contains any language feature, such as a keyword, published by any party, that is not defined or enabled by the Specifications.

4. The presence of a comment or directive in source code, when compiled into Executable Output by the Compiler, must not cause the functional programmatic behavior of a class or interface or application to vary, when executed by the Runtime, from the functional programmatic behavior of the class or interface or application in the absence of that comment or directive.
5. The Executable Output of the Compiler must be in class file format defined by the Specifications.
6. Annotation processors activated in each Product Configuration must be documented.
7. For each Technology that is included in the Product, there must be a Product Configuration that causes the Compiler to process annotations as required by that Technology.
8. For each Product Configuration that causes the Compiler to create Executable Output as well as to process annotations, and for each class or interface for which both the Compiler and the Reference Compiler produce Executable Output, the Executable Output generated by the Compiler must be functionally equivalent to the Executable Output generated by the Reference Compiler.

2.2.4 Rules for Products That Include a Development Kit

The following Rules, in addition to the "2.2.2 Rules for Java SE 6 Technology Products" on page 34 and "2.2.3 Rules for Compiler Products" on page 36, apply for each version of an operating system, software component, and hardware platform Documented as supporting software development.

2.2.4.1 Rules for Products That Include a Schema Compiler and Generator

These Rules, in addition to the "2.2.2 Rules for Java SE 6 Technology Products" on page 34 and "2.2.3 Rules for Compiler Products" on page 36, apply to all products that include a Development Kit.

1. In cases where Schema Compiler Output contains Executable Output, that Schema Compiler Output must execute properly on a Reference Runtime.

2. The Schema Compiler must not produce Schema Compiler Output from schema that do not conform to the W3C XML Schema recommendation [XSD Part 1][XSD Part 2].
    a. The Schema Compiler in non-default Product Configurations may produce Schema Compiler Output when an XML Schema contains constructs for which a binding has not been defined by the Specification.
    b. The Schema Compiler in non-default Product Configurations may produce Schema Compiler Output when a custom binding declaration is encountered.
3. The Schema Compiler Output of the Schema Compiler must be either in class format defined by the Java Virtual Machine Specifications or in source file format defined by Java Language Specification (JLS).
4. For each Product Configuration that causes the Schema Generator to produce Schema Generator Output, the Output of Schema Generator must fully meet W3C requirements for the XML Schema language.
5. The Schema Generator must not produce Schema Generator Output when a program element violates mapping constraints defined by the Specifications.
6. The Schema Generator may produce Schema Generator Output when a program element contains constructs for which a mapping has not been defined by the Specification.
7. A schema produced by the Schema Generator from source code containing a comment or a directive, or from a class file containing non-standard attributes, must be equivalent to a schema produced by the Schema Generator from the same source code with the comment or directive removed, or from the class file with the attributes removed. Two schemas are equivalent if and only if they validate the same set of XML documents (any XML document is either valid against both schemas or invalid against both schemas).

2.2.4.2 Rules for Products That Include Java-to-WSDL and WSDL-to-Java Tools

These Rules, in addition to the "2.2.2 Rules for Java SE 6 Technology Products" on page 34 and "2.2.3 Rules for Compiler Products" on page 36, apply to all products that include a Development Kit.

1. Source Code in WSDL-to-Java Output when compiled by a Reference Compiler must execute properly when run on a Reference Runtime.
2. Source Code in WSDL-to-Java Output must be in source file format defined by the Java Language specification (JLS).
3. Java-to-WSDL Output must fully meet W3C requirements for the Web Services Description language (WSDL) 1.1.
4. A Java-to-WSDL Tool must not produce Java-to-WSDL Output from source code that does not conform to the Java Language Specification (JLS).

2.3 JCK Test Appeals Process

Sun has a well-established process for managing challenges to its Java SE 6 technology Test Suite and plans to continue using a similar process in the future. Sun, as the Java SE 6 technology Maintenance Lead, will authorize representatives from the Java Licensee Engineering group to be the point of contact for all test challenges. Typically this is to be the engineer assigned to a company as part of its Java SE 6 technology JCK support.

If a test is determined to be invalid in function or if its basis in the specification is suspect, the test may be challenged by any licensee of the Java SE 6 technology JCK. Each test validity issue must be covered by a separate test challenge. Test validity or invalidity is to be determined based on its technical correctness such as:

- Test has bugs (i.e., program logic errors)
- Specification item covered by the test is ambiguous
- Test does not match the specification
- Test assumes unreasonable hardware and/or software requirements
- Test is biased to a particular implementation

Challenges based upon issues unrelated to technical correctness as defined by the specification will normally be rejected.

Test challenges must be made in writing to Java Licensee Engineering and include all relevant information as described in the "2.3.1 Test Challenge Form" on page 40. The process used to determine the validity or invalidity of a test (or related group of tests) is described in "JCK Test Appeals Steps" on page 40.

All tests found to be invalid will either be placed on the Exclude List for that version of the Java SE 6 technology JCK or have an alternate test made available.

- Tests that are placed on the Exclude List is to be placed on the Exclude List within one business day after the determination of test validity. The new Exclude List is to be made available to all Java SE 6 technology JCK licensees on the Java SE 6 technology JCK web site.
- Sun, as Maintenance Lead has the option of creating alternative tests to address any challenge. Alternative tests (and criteria for their use) is to be made available on the Java SE 6 technology JCK web site.

Note – Passing an alternative test is deemed equivalent to passing the original test.

▼ JCK Test Appeals Steps

1 **Java SE 6 technology JCK licensee writes a test challenge to Java Licensee Engineering contesting the validity of one or a related set of Java SE 6 technology tests.**

A detailed justification for why each test should be invalidated must be included with the challenge as described by the "2.3.1 Test Challenge Form" on page 40.

2 **Java Licensee Engineering evaluates the challenge.**

If the appeal is incomplete or unclear, it is returned to the submitting licensee for correction. If all is in order, Java Licensee Engineering will check with the responsible test developers to review the purpose and validity of the test before writing a response. Java Licensee Engineering will attempt to complete the response within 5 business days. If the challenge is similar to a previously rejected test challenge (i.e., same test and justification), Java Licensee Engineering will send the previous response to the licensee.

3 **The challenge and any supporting materials from test developers is sent to the specification engineers for evaluation.**

A decision of test validity or invalidity is normally made within 15 working days of receipt of the challenge. All decisions is to be documented with an explanation of why test validity was maintained or rejected.

4 **The licensee is informed of the decision and proceeds accordingly.**

If the test challenge is approved and one or more tests are invalidated, Sun places the tests on the Exclude List for that version of the Java SE 6 technology JCK (effectively removing the test(s) from the Test Suite). All tests placed on the Exclude List will have a bug report written to document the decision and made available to all licensees through the bug reporting database. If the test is valid but difficult to pass due to hardware or operating system limitations, Sun may choose to provide an alternate test to use in place of the original test (all alternate tests are made available to the licensee community).

5 **If the test challenge is rejected, the licensee may choose to escalate the decision to the Executive Committee (EC), however, it is expected that the licensee would continue to work with Sun to resolve the issue and only involve the EC as a last resort.**

2.3.1 Test Challenge Form

Provide the following information to Java Licensee Engineering.

```
Test Challenger Name and Company:
Specification Name(s) and Version(s):
Test Suite Name and Version:
Exclude List Version:
```

```
Test Name:
Complaint (argument for why test is invalid):
```

2.3.2 Test Challenge Response Form

The following information is provided in response to a test challenge.

```
Test Defender Name and Company:
Test Defender Role in Defense (e.g., test developer, Maintenance Lead, etc.):
Specification Name(s) and Version(s):
Test Suite Name and Version:
Test Name:
Defense (argument for why test is valid):
[Multiple challenges and corresponding responses may be listed here.]
Implications of test invalidity (e.g., other affected tests and test
framework code, creation or exposure of ambiguities in spec (due
to unspecified requirements), invalidation of the reference
implementation, creation of serious holes in test suite):
Alternatives (e.g., are alternate test(s) appropriate?):
```

2.4 Reference Compiler for Java SE 6 Technology

The designated Reference Compiler for compatibility testing of Java SE 6 technology implementations is the Sun `javac` compiler in JDK release 6 executed using a Reference Runtime.

2.5 Reference Runtime for Java SE 6 Technology

Designated Reference Runtimes for compatibility testing of Java SE 6 technology implementations are the Java SE 6 technology Java Runtime Environment (JRE™) for Solaris Operating System on SPARC® technology, and for Win32 on Windows 2000 and Windows XP.

2.6 Specifications for Java SE 6 Technology

The Specifications for Java SE 6 technology are found on the Java Licensee Engineering web site.

2.7 Libraries for Java SE 6 Technology

Table 2–2 provides a list of packages that constitute the required class libraries for the Java SE 6 technology.

TABLE 2–2 Required Class Libraries for Java SE 6 Technology

Package Name
java.applet
java.awt
java.awt.color
java.awt.datatransfer
java.awt.dnd
java.awt.event
java.awt.font
java.awt.geom
java.awt.im
java.awt.im.spi
java.awt.image
java.awt.image.renderable
java.awt.print
java.beans
java.beans.beancontext
java.io
java.lang
java.lang.annotation
java.lang.instrument
java.lang.management
java.lang.ref
java.lang.reflect
java.math
java.net

TABLE 2–2 Required Class Libraries for Java SE 6 Technology *(Continued)*

Package Name
java.nio
java.nio.channels
java.nio.channels.spi
java.nio.charset
java.nio.charset.spi
java.rmi
java.rmi.activation
java.rmi.dgc
java.rmi.registry
java.rmi.server
java.security
java.security.acl
java.security.cert
java.security.interfaces
java.security.spec
java.sql
java.text
java.text.spi
java.util
java.util.concurrent
java.util.concurrent.atomic
java.util.concurrent.locks
java.util.jar
java.util.logging
java.util.prefs
java.util.regex
java.util.spi
java.util.zip

TABLE 2–2 Required Class Libraries for Java SE 6 Technology *(Continued)*

Package Name
javax.accessibility
javax.activation
javax.activity
javax.annotation
javax.annotation.processing
javax.crypto
javax.crypto.interfaces
javax.crypto.spec
javax.imageio
javax.imageio.event
javax.imageio.metadata
javax.imageio.plugins.bmp
javax.imageio.plugins.jpeg
javax.imageio.spi
javax.imageio.stream
javax.jws
javax.jws.soap
javax.lang.model
javax.lang.model.element
javax.lang.model.type
javax.lang.model.util
javax.management
javax.management.loading
javax.management.modelmbean
javax.management.monitor
javax.management.openmbean
javax.management.relation
javax.management.remote

TABLE 2–2 Required Class Libraries for Java SE 6 Technology *(Continued)*

Package Name
javax.management.remote.rmi
javax.management.timer
javax.naming
javax.naming.directory
javax.naming.event
javax.naming.ldap
javax.naming.spi
javax.net
javax.net.ssl
javax.print
javax.print.attribute
javax.print.attribute.standard
javax.print.event
javax.rmi
javax.rmi.CORBA
javax.rmi.ssl
javax.script
javax.security.auth
javax.security.auth.callback
javax.security.auth.kerberos
javax.security.auth.login
javax.security.auth.spi
javax.security.auth.x500
javax.security.cert
javax.security.sasl
javax.sound.midi
javax.sound.midi.spi
javax.sound.sampled

TABLE 2–2 Required Class Libraries for Java SE 6 Technology *(Continued)*

Package Name
javax.sound.sampled.spi
javax.sql
javax.sql.rowset
javax.sql.rowset.serial
javax.sql.rowset.spi
javax.swing
javax.swing.border
javax.swing.colorchooser
javax.swing.event
javax.swing.filechooser
javax.swing.plaf
javax.swing.plaf.basic
javax.swing.plaf.metal
javax.swing.plaf.multi
javax.swing.plaf.synth
javax.swing.table
javax.swing.text
javax.swing.text.html
javax.swing.text.html.parser
javax.swing.text.rtf
javax.swing.tree
javax.swing.undo
javax.tools
javax.transaction
javax.transaction.xa
javax.xml
javax.xml.bind
javax.xml.bind.annotation

TABLE 2–2 Required Class Libraries for Java SE 6 Technology *(Continued)*

Package Name
javax.xml.bind.annotation.adapters
javax.xml.bind.attachment
javax.xml.bind.helpers
javax.xml.bind.util
javax.xml.crypto
javax.xml.crypto.dom
javax.xml.crypto.dsig
javax.xml.crypto.dsig.dom
javax.xml.crypto.dsig.keyinfo
javax.xml.crypto.dsig.spec
javax.xml.datatype
javax.xml.namespace
javax.xml.parsers
javax.xml.soap
javax.xml.stream
javax.xml.stream.events
javax.xml.stream.util
javax.xml.transform
javax.xml.transform.dom
javax.xml.transform.sax
javax.xml.transform.stax
javax.xml.transform.stream
javax.xml.validation
javax.xml.ws
javax.xml.ws.handler
javax.xml.ws.handler.soap
javax.xml.ws.http
javax.xml.ws.soap

TABLE 2–2 Required Class Libraries for Java SE 6 Technology *(Continued)*

Package Name
javax.xml.ws.spi
javax.xml.ws.wsaddressing[1]
javax.xml.xpath
org.ietf.jgss
org.omg.CORBA
org.omg.CORBA_2_3
org.omg.CORBA_2_3.portable
org.omg.CORBA.DynAnyPackage
org.omg.CORBA.ORBPackage
org.omg.CORBA.portable
org.omg.CORBA.TypeCodePackage
org.omg.CosNaming
org.omg.CosNaming.NamingContextExtPackage
org.omg.CosNaming.NamingContextPackage
org.omg.Dynamic
org.omg.DynamicAny
org.omg.DynamicAny.DynAnyFactoryPackage
org.omg.DynamicAny.DynAnyPackage
org.omg.IOP
org.omg.IOP.CodecFactoryPackage
org.omg.IOP.CodecPackage
org.omg.Messaging
org.omg.PortableInterceptor
org.omg.PortableInterceptor.ORBInitInfoPackage
org.omg.PortableServer
org.omg.PortableServer.CurrentPackage
org.omg.PortableServer.POAManagerPackage

[1] Applicable for JAX-WS 2.1.

TABLE 2–2 Required Class Libraries for Java SE 6 Technology *(Continued)*

Package Name
`org.omg.PortableServer.POAPackage`
`org.omg.PortableServer.portable`
`org.omg.PortableServer.ServantLocatorPackage`
`org.omg.SendingContext`
`org.omg.stub.java.rmi`
`org.w3c.dom`
`org.w3c.dom.bootstrap`
`org.w3c.dom.events`
`org.w3c.dom.ls`
`org.xml.sax`
`org.xml.sax.ext`
`org.xml.sax.helpers`

2.7.1 Endorsed Standards for Java SE 6 Technology

The endorsed standards for the Java SE 6 technology constitute all classes and interfaces that are defined in the packages listed in the following location:

`http://java.sun.com/javase/6/docs/technotes/guides/standards/index.html`

Classes and interfaces defined in the subpackages of listed packages are not endorsed standards unless those subpackages are themselves listed.

Sun, as specification maintenance lead, may add additional endorsed standards to this list, but will not remove any items. Modified versions of the Endorsed Standards list will be posted to `http://java.sun.com/javase/6/docs/technotes/guides/standards/index.html` and apply to all licensees.

◆ ◆ ◆ CHAPTER 3

Installation

This chapter contains instructions for installing the JCK 6b test suites.

3.1 Delivery

JCK 6b is delivered in three Java Archive (JAR™) files that can be found at the licensee web site. Each of the JAR files contains a full test suite (compiler, runtime, and devtools) and a complete set of JCK documentation and tools.

In addition to the full test suites, JCK 6b provides sample compiler, runtime, and devtools test suites to aid you in configuring and testing your system before running the full test suites. The sample test suites contain actual JCK 6b tests, but the number of included tests is significantly reduced.

3.2 Installing JCK Test Suites

The JCK 6b products are shipped in self-executing `.jar` files.

To install JCK 6b test suites, execute the `.jar` files from the command line. The JCK test suites can also be manually unpacked and installed.

If you are testing on a Win32 platform, use either a FAT32 or NTFS file system when possible.

Note – Be sure to use the corresponding JDK software for the JCK version that you are installing.

You *might* need to install JCK 6b on a stable platform and use the JavaTest harness agent to run the tests on your target platform. See Chapter 4 to determine if this is the required configuration for your test environment.

▼ Command Line Installation

- **Execute the following command:**

  `java -jar` *PATH-TO-JCK-JAR options* `[-o` *outputdir*`]`

  The following descriptions are provided for the installation command:

  - `java` — Java SE platform reference implementation.
  - *PATH-TO-JCK-JAR* — The path of the JCK 6b JAR file to install.
  - *options* — Unless a user specifies options, the default behavior installs JCK 6b in verbose mode (writes the file names it is installing to standard out) into the current directory. The installation utility accepts the following options:
    - `-q` — Installs JCK 6b in quiet mode.
    - `-l` — Prints a list to standard output of what would be installed without installing.
    - `-install shell_scripts` — Installs executable scripts (in UNIX shell format) that run tests from command line. One script is provided for each test.
    - `-onError` *mode* — Sets the mode of sensitivity that the installer has to errors.

      The `-onError` option instructs the installer on what to do when it encounters an error while installing JCK 6b. This option has three valid modes:

      - `ignore` — Instructs the installer to ignore any errors and continue installation.
      - `report` — Instructs the installer to continue with the installation but report any errors when the installation is completed.
      - `fail` — Instructs the installer to stop installation once an error is encountered.
    - `-usage` — Prints a usage message to standard out and exits (not listed as an option in the usage message).
    - `-help` — Prints a usage message to standard out and exits (not listed as an option in the usage message).
  - `[-o` *outputdir*`]` — Installs JCK 6b into the directory represented by *outputdir*.

▼ Manual Installation

1. **Unzip all the data `*.zip` files from the product JAR files into a temporary directory.**

   The product JAR files contain the following elements:

   - `./com/sun/jck/utils/installer` directory contains the Java class library used by the installer.
   - `./META-INF` directory contains the manifest file for the JAR file.
   - One or more data `*.zip` files contain the JCK product.

- Copyright file.

2 **Unzip the data `*.zip` files from the temporary directory into the directory in which you want the JCK 6b product to reside.**

3.2.1 Directories and Files

Table 3–1 describes the directories and files contained in each JCK 6b test suite.

TABLE 3–1 JCK Test Suite Files and Directories

Directory or File	Contents
jck/COPYRIGHT-javatest.html	Copyright for the JavaTest harness online documentation.
jck/COPYRIGHT-jck.html	Copyright for the JCK 6b online documentation.
jck/Readme-JCK.txt *jck*/Readme-JCK.html	Information required to install JCK 6b on a system.
jck/build.txt	JCK build information.
jck/copyright.txt	Copyright for the JCK 6b files.
jck/index.html	Links to the contents of the JCK 6b installation.
jck/releaseNotes-JCK.txt *jck*/releaseNotes-JCK.html	Latest information and any changes from the previous JCK release.
jck/testsuite.jtt	JavaTest harness resource information for the test suite.
jck/classes/	Class files for the JCK 6b tests.
jck/classes/javasoft/sqe/tests/	Precompiled test files for the JCK.
jck/classes/javasoft/sqe/	Class files for the JCK related test frameworks libraries.
jck/doc/	Documentation for tests and tools.
jck/doc/jck/	JCK 6b documentation.
jck/doc/javatest/	JavaTest harness documentation and tutorials.
jck/lib/	JCK `.jtx` and `javatest.jar` files.
jck/lib/extensions	JAR files that extend a platform being tested.
jck/lib/extensions/LocaleProviders.jar	Special locale providers for testing the Pluggable Locales functionality.

TABLE 3–1 JCK Test Suite Files and Directories *(Continued)*

Directory or File	Contents
jck/lib/javatest.jar	The JavaTest harness application JAR file. This file must be specified in the CLASSPATH shell environment variable to start the JavaTest harness.
jck/lib/jdk6b.jtx	List of excluded tests that licensees are not required to pass.
jck/linux/	JavaTest harness resource files used for running the harness on Linux platforms.
jck/linux/multiJVM-agent-Linux-RI.jti *jck*/linux/singleJVM-agent-Linux-RI.jti	Template files for configuring a test platform.
jck/solaris/	JavaTest harness resource files used for running the harness on Solaris platforms.
jck/solaris/multiJVM-agent-Solaris-RI.jti *jck*/solaris/singleJVM-agent-Solaris-RI.jti	Template files for configuring a test platform.
*jck*src/share/classes/javasoft/sqe/javatest/lib/	JavaTest harness script, command sources, and samples.
jck/src/share/classes/javasoft/sqe/jck/lib/	Sources for JCK 6b test framework libraries.
jck/tests/	Test program sources (.java, .jasm, .jcod) and test description (.html) files. Also referred to as RootDIR.
JCK-Runtime: ■ *jck*/tests/api ■ *jck*/tests/lang ■ *jck*/tests/vm ■ *jck*/tests/xml_schema	■ API tests ■ Language specification tests ■ Interpreter tests ■ Tests based on W3C XML Schema Test Suite ■ If you installed the scripts for running single tests, each test source folder will contain <test_name>.sh files.
JCK-Compiler: ■ *jck*/tests/api ■ *jck*/tests/lang	■ API tests ■ Language specification tests ■ If you installed the scripts for running single tests, each test source folder will contain <test_name>.sh files.
JCK-devtools: ■ *jck*/tests/java2schema ■ *jck*/tests/jaxws ■ *jck*/tests/schema2java ■ *jck*/tests/schema_bind ■ *jck*/tests/xml_schema	■ Java to Schema mapping tests ■ JAX-WS tests ■ Schema to Java mapping tests ■ Schema binding tests ■ Tests based on W3C XML Schema Test Suite

TABLE 3–1 JCK Test Suite Files and Directories *(Continued)*

Directory or File	Contents
jck/tests/testsuite.html	JCK 6b test suite root index page.
jck/win32/	JavaTest harness resource files used for running the harness on Windows platforms.
jck/win32/multiJVM-agent-WindowsXP-RI.jti *jck*/win32/singleJVM-agent-Windows-RI.jti	Template files for configuring a test platform.

3.3 Release Notes

Once you install JCK 6b, read the release notes (*jck*/releaseNotes-JCK.txt or *jck*/releaseNotes-JCK.html). This is the *most* current set of notes on the release and tracks the major changes between releases. This file contains last-minute changes or supplementary instructions.

Online versions of the release notes are provided in the JavaTest harness. This documentation can be displayed in the JavaTest harness by choosing it from the JavaTest harness Help menu.

3.4 Accessing User Documentation

JCK 6b user documentation is provided in three formats:

- **PDF** — Acrobat Reader is required to view the documentation in this format.
- **HTML** — Use any web browser to view the documentation in this format.
- **JavaHelp™ tool** — Use the JavaTest harness Help menu to display this documentation in the JavaTest harness.

◆ ◆ ◆ CHAPTER 4

Running JCK 6b Tests

JCK 6b tests work in a variety of computing environments and can be difficult to set up and run in your environment. To reduce the difficulty of setting up a test run, the JavaTest harness uses the Configuration Editor with the JCK 6b configuration interview provided by the test suite to produce the configuration information required to run the tests.

Running JCK 6b tests consists of the following basic operations:

- "4.1 Starting the JavaTest Harness" on page 57
- "4.2 Generating a Configuration" on page 61
- "4.3 Running Tests" on page 63

Refer to "4.4 Configuration Interview" on page 65 for a description of the basic structure of the configuration interviews provided by the JCK test suites.

In some cases, tests require special setup procedures before they can be run. Refer to "4.5 Special Setup Instructions" on page 104 for a list of these tests. In addition to special setup procedures, some tests require multiple runs using different configurations. Refer to "4.6 Special Test Execution Requirements" on page 106 for a list of these tests.

4.1 Starting the JavaTest Harness

You can use the JavaTest harness GUI or the JavaTest harness command-line interface to run tests, write test reports, and audit tests. While most tasks can be performed in either the JavaTest harness GUI or the command-line interface, some special tasks, such as creating configurations, can only be performed by using the JavaTest harness GUI.

Note – While running the tests under win32, `java.io` API test failures can occur if a working directory is kept on the remote drive. These failures occur when file caching functionality is included with the remote file system access services. To avoid the dependence on a network file access services, whenever possible keep your working directory on a local system where you run the JCK tests. If you see `java.io` test failures reporting errors (such as the data read is not that previously written or a file length is not the expected length), move the work directory to the local drive or use different software (or software configuration) to access the remote drives.

Run the JavaTest harness on a computer with a minimum of 256 megabytes of physical memory. Use an agent with the JavaTest harness to run test programs on platforms with limited amounts of memory.

If you are running a large test suite (10,000 or more tests), you must allocate a large amount of memory to the Java virtual machine (VM) to run the JavaTest harness properly. When running a large test suite, allocate 512 megabytes of memory to the VM machine by including `-Xmx512m` in the command string used to start the JavaTest harness.

Note – When allocating memory to the VM, do not exceed the actual amount of memory available on your system.

In addition to allocating memory to the VM heap, additional memory is required for the Java HotSpot virtual machine if you run tests in single-VM mode. Memory in the Java HotSpot virtual machine is organized into three generations: a young generation, an old generation, and a permanent generation. The permanent generation holds objects that the VM finds convenient to have the garbage collector manage, such as objects describing classes and methods, as well as the classes and methods themselves.

The default value for the maximum size of the permanent generation memory (`-XX:MaxPermSize=n`) is platform-dependent. For 32-bit platforms, use 128mb when running tests in single-VM mode. This value is set with the following initial and maximum settings:

```
-XX:PermSize=128m -XX:MaxPermSize=128m
```

For 64-bit platforms, use 256 m for permanent-generation memory when running tests in single-VM mode. This value is set with the following initial and maximum, settings:

```
-XX:PermSize=256m -XX:MaxPermSize=256m
```

4.1.1 Starting the JavaTest Harness GUI

To start the GUI, perform any one of the following actions:

- **Execute the `javatest.jar` file.**

If you have access to a command line, you can directly execute the `javatest.jar` file from any directory by including its path in the command.

On UNIX systems, use the following format:

```
java -Xmx512m -XX:PermSize=256m -XX:MaxPermSize=256m -jar jt-dir/javatest.jar
```

On Win32 systems, use the following format:

```
java -Xmx512m -XX:PermSize=128m -XX:MaxPermSize=128m -jar jt-dir\javatest.jar
```

Note – The default RI PermGen values are not large enough to perform a complete JCK run at one time. 64-bit platforms require 256m permanent generation memory, whereas 32-bit platforms require 128m permanent generation memory. Include the appropriate options in the command to run tests for your test platform.

In the command line, *jt-dir* represents the path to `javatest.jar`. In JCK, the `javatest.jar` file is located in the `lib` directory of the test suite. If you are in the directory where the `javatest.jar` file is located, do not include the path in the command line.

For example, you can use the following command to start the GUI if you are in the directory where the `javatest.jar` file is located and are running tests in single-VM mode on a 64-bit platform:

```
java -Xmx512m -XX:PermSize=256m -XX:MaxPermSize=256m -jar javatest.jar
```

When starting the GUI, you can include additional commands in the command line that specify the work directory and the configuration values used to start the GUI.

To display a list of commands that can be used to launch the JavaTest harness GUI, type the following in a command line:

```
java -jar javatest.jar -help
```

See the *JavaTest Harness User's Guide: Command-Line Interface* document included with this release for detailed descriptions of these commands.

- **Double-click a shortcut icon.**

  If you are using a GUI, your system might support double clicking the `javatest.jar` file shortcut icon to launch the harness.

Note – If you choose to create and use a shortcut icon to launch the GUI, verify that your system allocates adequate memory to the VM.

- **Run a startup script.**
- **Invoke the class directly.**

If you have access to a command line and the JAR file is on the classpath, you can directly invoke the class from the directory where you intend to create files and store test results.

```
java com.sun.javatest.tool.Main
```

When starting the GUI from a command line, you can use additional commands to specify work directory and configuration values. See the *JavaTest Harness User's Guide: Command-Line Interface* document included with this release for detailed information about using additional commands and options to specify configuration values in the command line.

4.1.2 Using the JavaTest Harness Command-Line Interface

If you use a configuration template or a configuration generated in the JavaTest harness GUI, you can perform many tasks (such as modifying configuration files, specifying configuration values, running tests, writing reports, and auditing tests) from the JavaTest harness command-line interface that are performed in the GUI. Refer to *the JavaTest Harness User's Guide: Command-Line Interface* document included with this release or to the JavaTest harness online help.

4.1.3 Using the JavaTest Harness Quick Start Wizard

The JavaTest harness includes a Quick Start wizard (wizard) that enables users to quickly create combinations of test suite, work directory, and configuration files and open them in properly configured Test Manager windows. The harness opens the wizard when one of the following conditions occur:

- The harness cannot restore an existing desktop.
- A user includes a command-line option in the command to start the GUI.
- A user opens a new Test Manager window.

When you start the JavaTest harness for the first time, or choose to start the JavaTest harness with a new desktop, the harness displays the wizard. The wizard allows you to either specify the test suite, work directory, and configuration loaded in the GUI or to continue work on a previous test run.

If you use a configuration template or load an incomplete configuration, the wizard opens the configuration editor for you to complete the configuration before using the JavaTest harness.

4.2 Generating a Configuration

The JavaTest harness uses the Configuration Editor and the test suite configuration interview to generate a configuration that contains the values required to run tests on your test system. See "4.4 Configuration Interview" on page 65 for a description of the configuration interview.

Each JCK test suite provides a configuration interview that only contains the configuration questions required to collect information for running the tests of that test suite. By filtering unnecessary configuration information, each interview simplifies the process of creating a valid configuration for your test environment.

After the configuration is complete, the JavaTest harness generates a Question Log that lists each question and answer in the configuration interview. You can open the Question Log from the Test Manager. See "4.4.5 Configuration Question Log" on page 104 for additional information.

The harness saves the completed configuration and uses it to perform all test runs associated with the work directory.

Note – To temporarily change configuration values, you can set the configuration values in the command line or in the command file used to start the JavaTest harness. Refer to the *JavaTest Harness User's Guide: Command-Line Interface* or the JavaTest harness online help.

See Appendix D for a basic description of using the JavaTest harness. Detailed information about the JavaTest harness can be found in the *JavaTest Harness User's Guide: Graphical User Interface* and the *JavaTest Harness User's Guide: Command-Line Interface* included with this release or in the JavaTest harness online help. The JavaTest harness online help provides an extensive search capability that you can use to locate specific words or phrases in the online documentation.

4.2.1 Configuration Templates

You can create configuration templates from complete or partially completed configurations. When creating new configurations, you can use a configuration template to substantially reduce the number of values that must be provided in the interview.

You can also use the Configuration Editor bookmarks feature to hide the questions in a configuration template that do not require changes to their values. Using the bookmarks feature can substantially simplify the process of completing a configuration interview. Refer to the *JavaTest Harness User's Guide: Graphical User Interface* or the JavaTest harness online help for information about creating and using configuration templates and bookmarks.

JCK is shipped with a set of Templates that allow you to significantly shorten the time for configuring a test platform for certain JavaTest modes by filling in all necessary parameters.

Note – Parameters that depend on your configuration are left blank and need to be set manually.

All shipped templates belong to Sun's reference implementations and, therefore, contain parameters specific to this JDK realization. Each of the Runtime, Compiler, and Devtools test suite contains 6 templates:

- Single VM With an Agent for Solaris, Linux, and Windows platforms
- Multiple VMs With an Agent for Solaris, Linux, and Windows platforms

The templates are located in the following subfolders in the JCK Test Suite installation folder:

- *jck*/solaris/multiJVM-agent-Solaris-RI.jti
- *jck*/solaris/singleJVM-agent-Solaris-RI.jti
- *jck*/linux/multiJVM-agent-Linux-RI.jti
- *jck*/linux/singleJVM-agent-Linux-RI.jti
- *jck*/win32/multiJVM-agent-WindowsXP-RI.jti
- *jck*/win32/singleJVM-agent-Windows-RI.jti

4.2.1.1 Using a Template

The following procedures describe how to load template files.

▼ Loading a template in the Quick Start wizard

1. **In JavaTest Harness click** `File` > `Open Quick Start Wizard` **on the main menu.**
2. **Select Start a new test run and click Next.**
3. **Select Use a configuration template and specify the path to the template file or click Browse to locate the template. Click Next to continue.**

▼ Loading a template in the Configuration Editor

1. **Run Configuration Editor.**
2. **In the configuration editor click** `File` > `Load` **on the main menu**

4.2.2 Using Sample Test Suites

Sample test suites are provided with the test suite for you to use in creating and testing a configuration before you attempt to run the full JCK test suite. The `sampleJCK-compiler-6b`,

`sampleJCK-runtime-6b`, and `sampleJCK-devtools-6b` test suites contain samples of the different types of tests in the full JCK test suite. To use the sample test suites, install one of the sample test suite files.

After you start the JavaTest harness, load the `sampleJCK-compiler-6b`, `sampleJCK-runtime-6b`, or `sampleJCK-devtools-6b` test suite, and then complete the appropriate configuration. After you complete the configuration, test it by running the tests in the sample test suite. When you are able to successfully run the tests, you can use this configuration to run the full JCK 6b test suite. See "4.3 Running Tests" on page 63 for detailed description of using the JavaTest harness GUI to run tests.

See the *JavaTest Harness User's Guide: Graphical User Interface* included with this release or the JavaTest harness online help for detailed instructions about configuring and running tests.

4.3 Running Tests

You can run the JCK test suites either from the JavaTest harness GUI or from the command-line interface. If you are running interactive tests you must use the GUI. If you want to run the tests non-interactively, for example as part of a product build process, use the JavaTest harness command-line interface.

▼ Using the JavaTest Harness GUI

1 **Complete special test setup requirements.**

See "4.5 Special Setup Instructions" on page 104.

2 **Start the JavaTest harness.**

See "4.1 Starting the JavaTest Harness" on page 57.

3 **Choose File -> Open Quick Start wizard from the Test Manager menu bar if the harness does not open the wizard when it launches the GUI.**

4 **Choose one of the following options in the wizard and complete the associated questions:**

You are prompted to select a test suite.

a. **Start a New Test Run.**

b. **Resume Work on a Test Run.**

See "4.4 Configuration Interview" on page 65 for a description of the structure of each interview provided by each JCK test suite.

See the *JavaTest Harness User's Guide: Graphical User Interface* included with this release or the JavaTest harness online help for detailed instructions about using the configuration editor window.

5 **Click the Start Running Tests button on the Test Manager tool bar or choose Run Tests -> Start from the Test Manager menu bar.**

See the *JavaTest Harness User's Guide: Graphical User Interface* included with this release or the JavaTest harness online help for detailed instructions about using the Test Manager to run tests.

4.3.1 Using the Command-Line Interface

To run tests from the command line, you must use a configuration file created by the Configuration Editor. The JavaTest harness command-line interface provides a number of commands that you can use to specify individual configuration values used to run tests. However, values set in the command line override but do not change the values in the configuration file. You must use the Configuration Editor to make permanent changes in the configuration file.

See the *JavaTest Harness User's Guide: Command Line Interface* included with this release or the JavaTest harness online help for the commands and detailed instructions required to run tests from the command-line interface.

4.3.2 Setting Configuration Values in the Command Line

If you intend to set one or more configuration values in the command line for a test run, you must use one of the following sources to identify the appropriate configuration question-tag name:

- The Question Log file (`questionLog`*nnnn*`.html`) generated by the Configuration Editor in *workdirectory*`/jtData`.
- The `config.html` report generated from the JavaTest harness GUI.
- The Question Tags displayed in the Configuration Editor.

See the *JavaTest Harness User's Guide: Graphical User Interface* or the JavaTest harness online help for instructions required display the Question Tags in the Configuration Editor or generate reports containing the `config.html` file.

Note – The `questionLog`*nnnn*`.html` and `config.html` files can be viewed in any web browser.

4.4 Configuration Interview

The JCK 6b test suites include configuration interviews used by the JavaTest harness to collect information from you about your test environment and to create the configurations used to run tests. The configuration interviews use the More Info panel of the Configuration Editor to provide specific configuration information based on your answers to previous interview questions. See the *JavaTest Harness User's Guide: Graphical User Interface* included with this release or the JavaTest harness online help for detailed instructions about using the Configuration Editor.

A configuration interview consists of the following sequence of question categories:

1. **Configuration Modes and Tests to Run Questions** — The first section of each interview collects information about the mode in which you want to run the configuration interview and about which tests to run. See "4.4.1 Configuration Modes" on page 65 and "4.4.1.2 Specify Tests to Run" on page 71 for additional information.
2. **Test Environment Questions** — This section of an interview collects information about the test environment used to run the JCK 6b test suite (such as the type of environment and the type of agent used). See "4.4.2.1 JCK Runtime Test Suite" on page 72, "4.4.2.2 JCK Compiler Test Suite" on page 82, "4.4.2.3 JCK Devtools Test Suite" on page 92, and "4.4.2.4 Using Agents" on page 102 for additional information.

Note – The test environment related questions are displayed only if you run the tests and the interview in Advanced mode. See "4.4.1 Configuration Modes" on page 65 for more information on Advanced mode.

---

3. **Test-Specific Questions** — This section of each interview collects information about the tests from the test suite that is to be run. See Chapter 5 for detailed information.
4. **Test Run Questions** — The last section of each interview collects information about how the tests are run (such as Keywords and the exclude list) and contains test suite specific information and values. See "4.4.4.2 Keywords" on page 103 and "4.4.4.1 Exclude Lists" on page 103 for additional information.

4.4.1 Configuration Modes

You can run each configuration interview in one of the following modes:

- **Advanced** — Advanced mode is designed for use by expert users. This mode provides users with access to all configuration features including advanced features. In this mode the interview may contain over a hundred questions.
- **Simple** — Simple mode is designed for use by non-expert users. The number of interview questions is significantly reduced due to limited usage of advanced features and extensive usage of default values. In Simple mode it is assumed that the tests are run as follows:

- locally that is on the same machine as the interview
- without test agents
- each test is started in a separate virtual machine (VM)

If these limitations are not acceptable for a particular test run, then Advanced mode must be used. The number of interview questions is also reduced when only selected tests need to be run. In this case the interview questions that are not relevant to the selected tests are not displayed.

In Simple mode some of the default values are set based on the Sun reference implementation (RI) parameters. For example, in Simple mode the question about the syntax of the VM classpath option is not asked assuming that the syntax is the same as in the Sun RI. See Table 4–2 for more details. If the VM under test uses a different option syntax, then you can do one of the following:

- use a wrapper script to start the VM under test that emulates the command line interface of the Sun RI
- use Advanced mode that asks a set of questions about the syntax of options of the VM under test

---

Tip – In Simple mode, the **Test Run Questions** section of the configuration is hidden. You can access these values via the JavaTest GUI using the Quick Set Mode interview.

Tip – If you are in doubt which mode is appropriate for you, try Simple mode first. You can later switch the interview to Advanced mode and redefine standard values.

Caution – Before switching the interview from Advanced mode to Simple mode, save your configuration options because in Simple mode a number of hidden configuration settings will be initialized with default values and the answers that you already provided in Advanced mode may be lost.

Both Advanced and Simple modes can be used to certify a Java SE 6 implementation as described in Chapter 2, “Procedure for Java SE 6 Technology Certification.”

In the interview, configure the mode to run the tests by providing the answers from Table 4–1.

TABLE 4–1 Interview Answers That Configure Configuration Mode

Question	Answer
Advanced Mode — Do you wish to use the Advanced mode?	`Yes` — To run the interview in Advanced mode `No` — To run the interview in Simple mode

4.4.1.1 Simple Mode Default Values

In Simple mode a number of interview questions are hidden because they use answers initialized with default values.

Table 4–2 lists the default values that are set in Simple mode for the JCK Runtime test suite. The questions are listed in the order in which they appear in Advanced mode.

TABLE 4–2 Interview Answers — Default Values in Simple Mode for JCK Runtime Test Suite

Question	Default Value	Comments
Configuration Name — Provide a short identifier that names the configuration you are creating.	`jck_${product}_${OS}`	Simple mode assumption.
Description — Provide a short description that identifies the configuration you are creating.	None	Simple mode assumption.
Native Code — Does your system support direct access to native code?	`Yes`	Simple mode assumption.
JVM TI Support — Is the Java virtual machine Tool Interface (JVM TI) implementation provided?	`Yes`	Simple mode assumption.
Fallback Support — Have you configured the product VM in this test run to use the fallback mode (the type inference strategy) of class verification when the typechecking strategy fails?	`Yes`	Simple mode assumption. Fall back to the older verification when typechecking fails.
Headless System — Is the system you are testing headless?	`No`	Simple mode assumption. Display is available.
Test Execution Mode — Select the test execution mode	`MultiJVM`	Simple mode assumption.
Testing Locally — Are you running these tests on the same system used for this interview?	`Yes`	Simple mode assumption. Testing locally. No JavaTest agent is used.

TABLE 4–2 Interview Answers — Default Values in Simple Mode for JCK Runtime Test Suite *(Continued)*

Question	Default Value	Comments
Runtime Classpath — For the Java launcher that runs the tests, is the classpath specified with an environment variable or a command-line option?	`command line option`	Simple mode assumption.
Classpath Option — Enter an option template for use in setting the classpath for the Java launcher that runs the tests.	`-classpath #`	The same parameter as in the Sun RI.
Location of native libraries — Are all of the native libraries in one location?	`Yes`	Simple mode assumption.
Native Library Path Value — What is the name of the directory that contains the libraries of native code?	`PATH` (for Windows) or `LD_LIBRARY_PATH` (for Unix, Linux, and other)	The same parameter as in the Sun RI.
JVM TI Agent Option String — Enter an option template for use in specifying the JVM TI agent library. Use the character '#' in the option template to indicate where the library name is specified. Use the character '@' to indicate where the agent options string is specified.	`-agentlib:#=@`	The same parameter as in the Sun RI.
JVM TI Agent start-up in Live phase — Is JVM TI Agent start-up in Live phase supported?	`Yes`	Simple mode assumption.
JVM TI AttachConnector implementation — Specify the fully-qualified class name used for starting native agent in the VM during the live phase.	`javasoft.sqe.jck.lib.attach.` `JVMTIAttachConnector`	Other assumptions.
JAX-WS Version— What version of JAX-WS technology does your implementation support.	JAX-WS 2.1	
Std Activation Port — Use the standard RMI activation port for the RMI daemon?	`Yes`	The same parameter as in the Sun RI.
System Property Specification — Does your platform use the standard method of specifying system properties on the command line?	`–Dname=value`	The same parameter as in the Sun RI.

TABLE 4–2 Interview Answers — Default Values in Simple Mode for JCK Runtime Test Suite *(Continued)*

Question	Default Value	Comments
DISPLAY Variable — Some tests might require that you set the "DISPLAY" environment variable. Enter a value for the DISPLAY variable for the system on which you run the tests.	`DISPLAY` OS environment or `localhost:0.0`	The same parameter as in the Sun RI.
Verifier Option — JCK requires that the verifier is fully enabled for each test that is run. Specify all options that might be required by your Java launcher to enable the verifier.	`–Xfuture`	The same parameter as in the Sun RI.
Can Play Audio — Does your system provide support for producing audio?	`Yes`	Other assumptions.
Use the Provided Audio File — Does your system support WAV, AU or SND audio formats for playback?	`Yes`	Other assumptions. `.WAV`
Security Authorization Policy — Which of the following mechanisms does your product use to specify the Java Security Authorization Policy?	standard system property	The same parameter as in the Sun RI.
Security Login Configuration — Which of the following mechanisms does your product use to specify the Java Security Login Configuration?	standard system property	The same parameter as in the Sun RI.
Expect OutOfMemoryError — Can OutOfMemoryError be thrown on the system under test when memory is allocated?	`Yes`	Simple mode assumption.
Restrict TCP Ports — Does your operating system restrict the set of TCP ports available to tests that request a specific port?	`No`	Simple mode assumption.
Local Host Name — What is the host name of the system on which you run the tests?	`localhost`	Other assumptions.
Local Host IP Address — What is the IP address of the system on which you run the tests?	`127.0.0.1`	Other assumptions.

TABLE 4–2 Interview Answers — Default Values in Simple Mode for JCK Runtime Test Suite *(Continued)*

Question	Default Value	Comments
Restrict UDP Ports — Does your operating system restrict the set of UDP ports available to tests that request a specific port?	`No`	Simple mode assumption.
Need Proxy — Do you use a proxy to access the HTTP and FTP URLs you have just entered?	`No`	Simple mode assumption.
ORB Host — What is the name of the machine on which the Java IDL name server is run?	`localhost`	Other assumptions.
ORB Port — What is the port on which the Transient Name Service listens?	`900`	Other assumptions.
Printer — Is your system connected to a printer that is online?	`Yes`	Simple mode assumption.
AWT Robot — Does your system provide support for automated event generation, as defined by the java.awt.robot classes?	`Yes`	Simple mode assumption.
JDWP Support— Is the Java Debugger Wire Protocol (JDWP) implementation provided?	`Yes`	Simple mode assumption.
Transport Class Name — Specify the fully-qualified class name used for establishing connection between the tests debuggers and JDWP implementation.	`javasoft.sqe.jck.lib.jpda.jdwp.SocketTransportService`	Other assumptions.
Attaching or Listening debugger — Should the debugger listen for connections from or attach to the JDWP implementation?	attaching	Other assumptions.
Connection Address for Debugger — Specify the address for the debugger to use when listening for connections from or attaching to the JDWP implementation.	Passive agent host (`${RuntimeRemoteAgentInterview.passiveHost}`) or `localhost` as host and `35000` as port	Other assumptions.
VM Suspended On Start — Does the specified JDWP implementation configuration provide that the Java runtime is suspended on-start?	`Yes`	Simple mode assumption.

TABLE 4–2 Interview Answers — Default Values in Simple Mode for JCK Runtime Test Suite *(Continued)*

Question	Default Value	Comments
JDWP Command-line Options — Specify the command-line options for activating JDWP.	`-agentlib:jdwp=server=y,` `transport=dt_socket,` `address=${RuntimeJDWPInterview.` `transportAddress},suspend=Y`	The same parameter as in the Sun RI.
RowSet Factory — Enter the name of the factory class for creating the rowset implementations.	`javasoft.sqe.tests.api.javax.sql.` `rowset.impl.RowSetFactoryImpl`	Other assumptions.
JPLIS Agent start-up in Live phase — Does your runtime supports instrumentation agent start-up after the JVM is launched?	`Yes`	Simple mode assumption.
JPLIS AttachConnector implementation — Specify the fully-qualified class name used for starting instrumentation agent sometime after the JVM is launched.	`javasoft.sqe.jck.lib.attach.` `JPLISAttachConnector`	Other assumptions.

4.4.1.2 Specify Tests to Run

You can specify the tests that are run either in the command line or in the Configuration Editor. If you specify a test folder, all tests located hierarchically beneath it are automatically selected for you. See the *JavaTest Harness User's Guide: Graphical User Interface*, the *JavaTest Harness User's Guide: Command-Line Interface*, or the JavaTest harness online help for a detailed description of how to specify the tests that are run.

In the interview, select the tests to run by providing the answers from Table 4–3.

TABLE 4–3 Interview Answers That Configure The Tests to Run

Question	Answer
Specify Tests to Run — Do you wish to run only selected sections of the test suite?	`Yes` — To specify the tests to run `No` — To run all tests

4.4.2 Test Environment Questions

Note – The configuration interview displays test environment related questions only if you run the tests and the interview in Advanced mode. See "4.4.1 Configuration Modes" on page 65 for more information on Advanced mode.

The JCK 6b test suites can be run in many different test environments. The first section of each interview contains a series of sub-interviews that collect the information required to configure a test environment.

To successfully pass the tests in the JCK 6b test suites, you must set the following options in the Test Execution section of each interview:

- Use the Verifier Option question to enable full class file verification in the configuration used to run the tests.

  To enable full class file verification, set the `-Xfuture` option in the Verifier Option field of the JCK interview.
- When you select Yes in the Test Platform MultiJVM question, use the Other Options question to specify the extension directory that extends the tested platform for testing the pluggability of locales provided by an implementation.

  Note – To test the pluggability of locales provided by an implementation, a tested platform must be extended by the JAR files in *jck*`/lib/extensions` each time it is started.

  For Sun implementations, set the following option in the Other Options field:

  `-Djava.ext.dirs=`*jck*`/lib/extensions:` *java-home*`/lib/ext:`*jre-home*`/lib/ext`

  If you do not select Yes in the Test Platform MultiJVM question you cannot use the Other Options question to specify the extension directory. When you start an agent (from a command line or a browser) to run tests, specify *jck*`/lib/extensions` as an additional extension directory.

4.4.2.1 JCK Runtime Test Suite

The interview presents a series of questions and uses their answers to create a configuration for the test environment of the JCK 6b runtime test suite. The following sections describe the test environments that might be configured for the test run:

- "Multiple VMs Without an Agent" on page 73
- "Multiple VMs With an Agent" on page 74
- "Multiple VMs Group With an Agent" on page 78
- "Multiple VMs Group Without an Agent" on page 75
- "Single VM With an Agent" on page 79

Note – The illustrations and descriptions in the following sections describe the interactions of the various VMs and systems involved in running the tests. In addition, some tests require the use of other servers, possibly on additional systems. For simplicity these servers and systems are not included here. These remote systems are described in "4.4.2.1 JCK Runtime Test Suite" on page 72 and in Chapter 5.

Multiple VMs Without an Agent

In the interview, when you provide the answers from Table 4–4 or from Table 4–5, the Configuration Editor creates a configuration that uses multiple VMs without an agent to run the runtime tests.

TABLE 4–4 Interview Answers (Runtime Running Locally) That Configure Multiple VMs Without an Agent

Question	Answer
Test Execution Mode — Select the test execution mode	**MultiJVM**
Testing Locally — Are you running these tests on the same system used for this interview?	`Yes`

TABLE 4–5 Interview Answers (Runtime Running Remotely) That Configure Multiple VMs Without an Agent

Question	Answer
Test Execution Mode — Select the test execution mode	**MultiJVM**
Testing Locally — Are you running these tests on the same system used for this interview?	`No`
Test Platform — Which operating system is used on the test platform?	Any choice but `Other`

In the test environment illustrated in Figure 4–1, the JavaTest harness and the JCK tests run in separate Java technology runtimes. The harness starts a new VM for every test. This test environment does not use an agent.

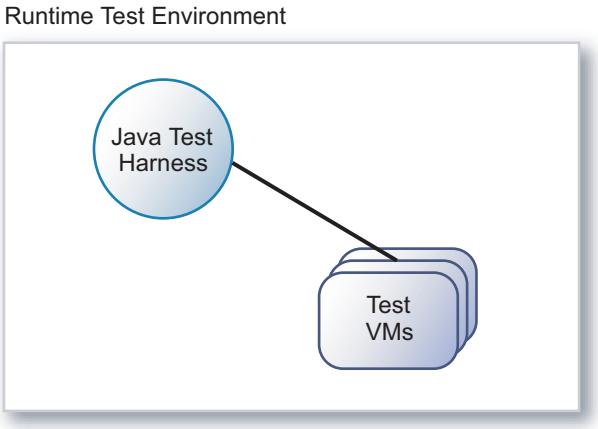


FIGURE 4–1 Runtime Test Environment - Multiple VMs Without an Agent

---

Note – If the harness displays errors indicating that classes are not found, check that the classpaths were set correctly when configuring the classpath for each of the VMs.

Multiple VMs With an Agent

In the interview, when you provide the answers to questions listed in Table 4–6, the Configuration Editor creates a configuration that uses multiple VMs with an agent for running the runtime tests.

TABLE 4–6 Interview Answers That Configure Multiple VMs With an Agent

Question	Answer
Test Execution Mode — Select the test execution mode	**MultiJVM**
Testing Locally — Are you running these tests on the same system used for this interview?	`No`
Test Platform — Which operating system is used on the test platform?	`Other`

In the test environment illustrated in Figure 4–2, the JavaTest harness runs on a separate computer (using one of the reference VMs) while an agent runs the tests in separate VMs on the system under test. The agent, JCK tests, and the runtime product run in separate Java runtime environments on the target system. The harness starts a new VM for every test. See "4.4.2.4 Using Agents" on page 102 for a description of the types of agents that you can use.

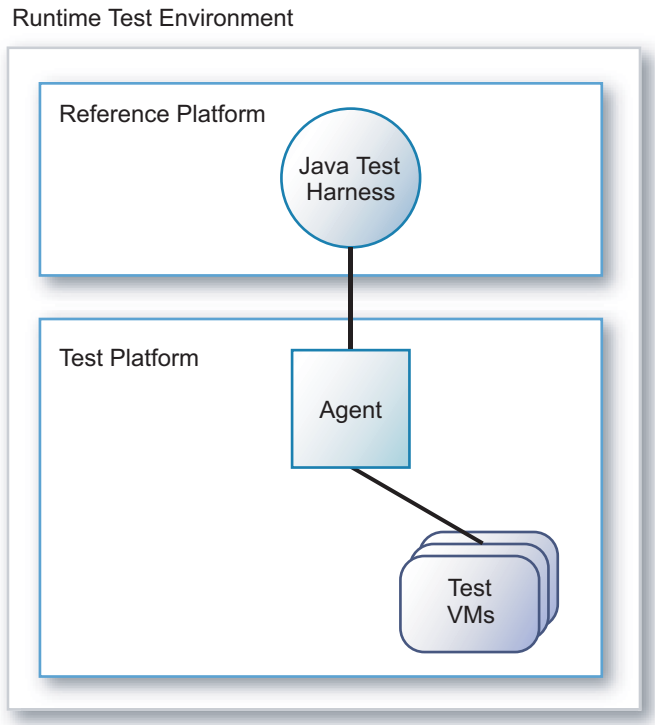


FIGURE 4–2 Runtime Test Environment — Multiple VMs With an Agent

Note – If the harness displays errors indicating that classes are not found, check that the class paths were set correctly when configuring the classpath for each of the VMs.

Multiple VMs Group Without an Agent

TABLE 4–7 Interview Answers That Configure Multiple VMs Group Without an Agent

Question	Answer
Test Execution Mode — Select the test execution mode	**MultiJVM group**
Testing Locally — Are you running these tests on the same system used for this interview?	**Yes**

MutliJVM group mode is intended to reduce the overhead of starting VMs by sharing VM instances among groups of test executions. In this mode the harness starts an internal agent VM on the system that JavaTest is run on and passes a group of the test execution to this agent. Once

the internal agent completes the test group run, it closes. The new internal agent VM is started for the next group of tests. The JavaTest harness starts socket server in particular port to communicate with internal agents running on the same system. By default, the port number is chosen from the list of available ports but you can override this parameter.

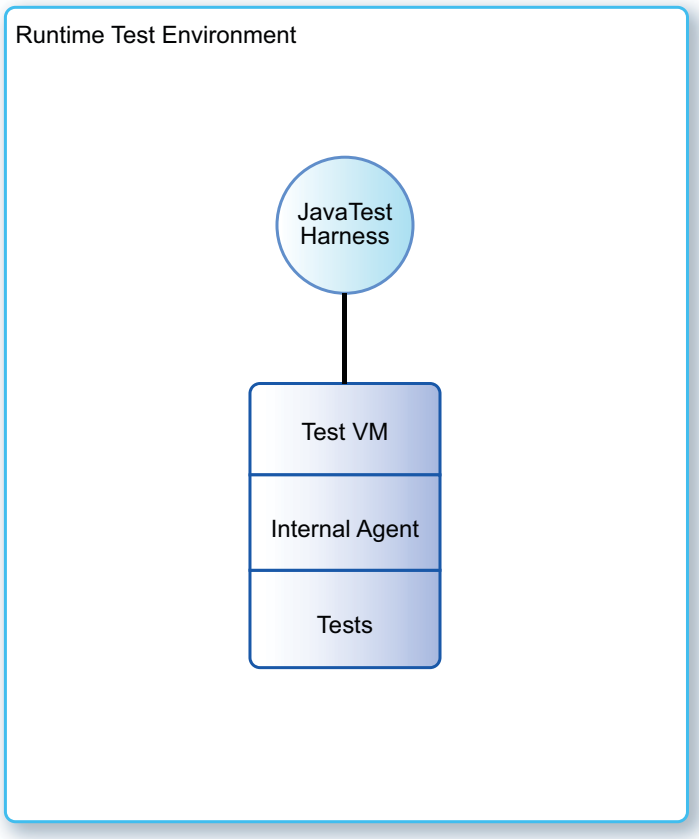


FIGURE 4–3 Runtime Test Environment — Multiple VMs Group Without an Agent

---

Note – If you have a firewall installed on your system, make sure it allows incoming connections from the local host.

The internal agent waits for a test execution request from the JavaTest harness. If no test request received during a timeout, the agent VM is automatically closed.

The configuration of the **MutliJVM group** test execution is the same as the configuration of the **MutliJVM** mode except for several parameters listed in Table 4–8 that control the **MutliJVM group** mode execution.

Note – These parameters are not visible in the JCK configuration interview and can only be changed by setting corresponding test environments variables via the JavaTest harness command line interface. For example:

```
javatest -jar lib/javatest.jar
-Ejck.env.runtime.testExecute.groupMode.serverPort=8080
```

This variable sets TCP port used for listening for incoming connections from internal agents.

See Additional Setup Commands chapter in the JavaTest Harness Command-Line Interface User's Guide.

TABLE 4–8 Environment variables that control execution of the MutliJVM group mode

Variable	Description
jck.env.runtime.testExecute.groupMode.groupSize	Controls a number of tests executed as a group in the same agent VM instance started by the JavaTest harness. Setting groupSize value to *>1* turns the group mode on. Setting groupSize to *1* turns the group mode off. Setting groupSize value to *0* removes limitation on the number of tests executed by one VM (Default is 100). Default group size provides optimal balance between performance and stability of JCK runs.
jck.env.runtime.testExecute.groupMode.serverPort	A TCP port used for listening for incoming connections from internal agents. Default value is 0, which means that JVM will automatically choose a free port on the test system. You can specify a particular port number to avoid firewall issues.
jck.env.runtime.testExecute.groupMode.timeout	Timeout (in seconds) between test executions during which an agent is active. Default is 60 seconds. Increase this value if necessary.
jck.env.runtime.testExecute.groupMode.agentArgs	Additional agent command line arguments passed to internal agents while starting. For example, you can specify "-trace" option to enable diagnostic output from the agent.
jck.env.runtime.testExecute.groupMode.closeIfFail	A policy used to close agents after test execution if a test status is fail. Default is false, i.e. unset.

TABLE 4–8 Environment variables that control execution of the MutliJVM group mode *(Continued)*

jck.env.runtime.testExecute.groupMode.keepOnError	A policy used to close agents after test execution if a test status is error. Default is false, i.e. unset.
jck.env.runtime.testExecute.groupMode.debug	Setting this variable enables additional logging. By default debug is off.

Note – When you select the **MutliJVM group** mode, JCK does its best to optimize execution by grouping tests and executing them in the same VM instance. However, in case tests cannot be grouped (for example, tests marked by the following keywords: "interactive", "jdwp", "jplis", "jvmti", "jplislivephase", "jvmtilivephase", "jniinvocationapi", "only_once"), the tests will be executed individually.

Multiple VMs Group With an Agent

TABLE 4–9 Interview answers that configure Multiple VMs group with an agent

Question	Answer
Test Execution Mode — Select the test execution mode	**MultiJVM group**
Testing Locally — Are you running these tests on the same system used for this interview?	**No**
Test Platform	**Other**

When you answer interview questions as specified in Table 4–9, the JavaTest harness runs on a separate computer in the test environment (using one of the reference VMs) while an agent runs tests in separate VMs system under test. The agent, JCK tests, and the runtime product run in separate Java runtime environments on the target system. The agent starts a new internal agent VM on the target system and passes a group of test execution to this agent.

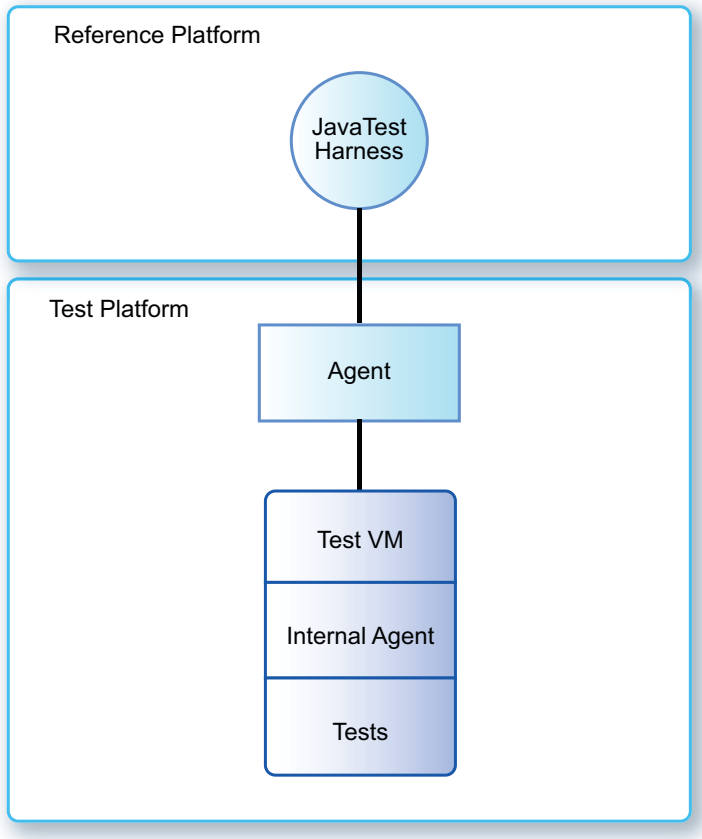


FIGURE 4–4 Runtime Test Environment — Multiple VMs Group With an Agent

All **MultiJVM group** mode configuration parameters, mentioned in Table 4–8 apply to this execution mode.

See "4.4.2.4 Using Agents" on page 102 for a description of the types of agents that you can use.

Single VM With an Agent

In the interview, when you select **SingleJVM** in the following question, the Configuration Editor creates a configuration using a single VM with an agent to run the runtime tests:

Test Execution Mode — Select the test execution mode

In the test environment illustrated in Figure 4–5, the JavaTest harness runs on a separate computer (using one of the Sun reference VMs), while an agent and the JCK tests run on your runtime product located on the target system.

This test environment is used when the system under test does not support multiple processes. Because the agent and the JCK tests run in the same VM, the classpath for the agent VM must be correct for both the agent and the tests. See "4.4.2.4 Using Agents" on page 102 for a description of the types of agents that you can use.

Note – If the harness displays errors indicating that classes are not found, check that the class paths were set correctly when configuring the class path for each of the VMs.

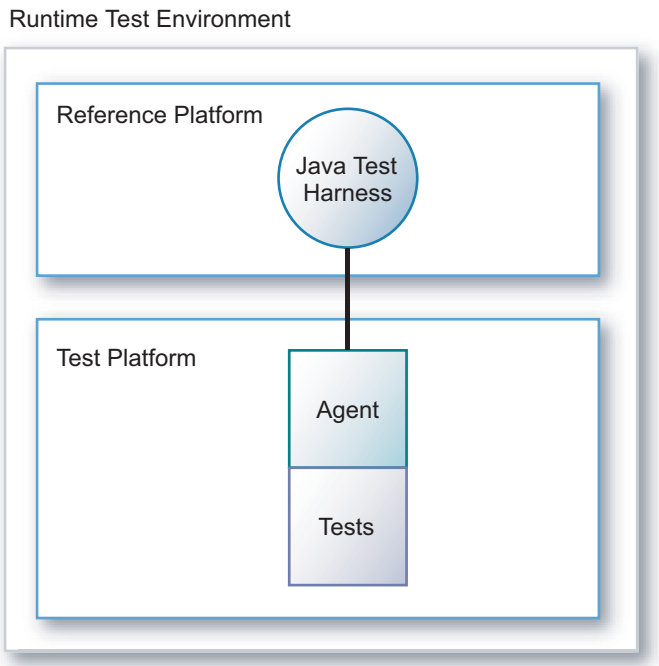


FIGURE 4–5 Runtime Test Environment — Single VM With an Agent

If your product runs in a browser on Win32 or Solaris platform, Figure 4–6 can also represent your test configuration. The dashed box in Figure 4–6 represents the browser. In the test environment illustrated in Figure 4–6, the JavaTest harness, agent and the tests run in a browser on the same system (the runtime product).

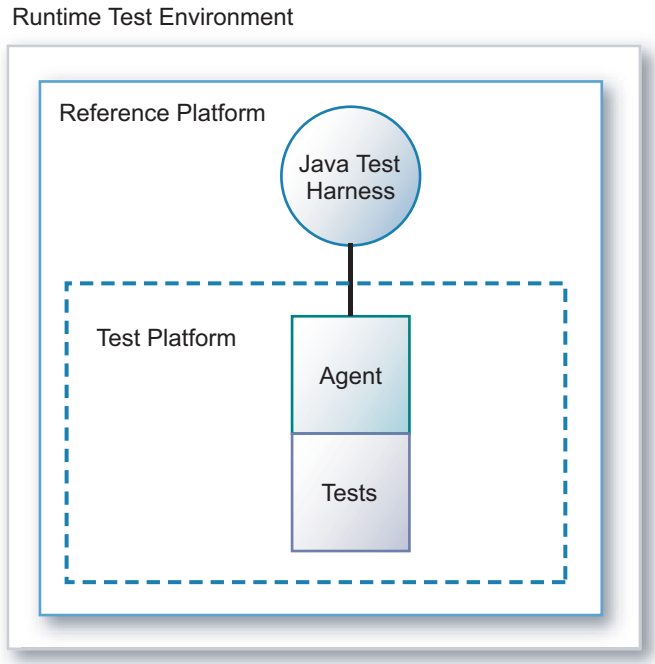


FIGURE 4–6 Runtime Test Environment — Single VM Browser Environment

A significant difference exists between the single and multiple VM environment configurations. In multiple VM environment configuration, *all* of the information required to configure and execute the Java runtime environment under test is specified in the JCK configuration file. In single VM environments, configuration information is spread across the following multiple locations:

- In the current JCK configuration
- On the command line for the Java runtime environment under test
- In the HTML file used to load the agent applet (if you are testing a browser)

Because the Java runtime environment under test is only launched once for a test run, the bulk of the JCK configuration happens when the VM is started.

- The Java technology security manager must be correctly configured to run tests.
- The native code library path must be configured to allow the Java Native Interface (JNI™) implementation and Extra-Attribute tests to run.
- Any necessary VM options must be specified.
- Any other required shell environment variables must be specified before the runtime under test is started with the agent.

- If you are *not* testing a browser and not using the `-loadDir` option, the class path must also be correctly specified to include the agent and the test classes before starting the runtime under test.
- If you are testing a browser implementation of the Java runtime environment, you must supply the correct classpath in an HTML file.

4.4.2.2 JCK Compiler Test Suite

The interview presents a series of questions and uses their answers to create a configuration for the test environment of the JCK 6b compiler test suite. The following sections describe the possible configurations of test environments used to run the tests of the JCK compiler test suite:

- "Multiple VMs Without an Agent" on page 82
- "Multiple VMs With an Agent" on page 83
- "Multiple VMs Group Without an Agent" on page 85
- "Multiple VMs Group With an Agent" on page 89
- "Single VM With an Agent" on page 90

It is important to note how the JCK compiler tests work. RMI compilers and the Java programming language compiler (Java compiler) compile test files and the resulting class files are executed on a Sun reference runtime implementation. To compile and execute compiler tests, you must configure five test commands that invoke the following compilers and runtime:

- Java complier
- RMI compiler
- RMI Compiler v1.1
- RMI Compiler v1.2
- Sun reference runtime

Table 4–10 describes the different types of compiler tests.

TABLE 4–10 Types of JCK Compiler Tests

Type of Test	Processes Started
Negative	Only the Java compiler is started, the compiler is expected to fail and produce an error. A new compiler is started for every test.
Positive (Java compiler)	A new Java compiler is started for every test. Output of the compiler is always checked using a new reference VM.
Positive (RMI compiler)	New Java and RMI compilers are started for every test. Output of the compilers is always checked using a new reference VM.

Multiple VMs Without an Agent

In the interview, when you provide the answers from Table 4–11, the Configuration Editor creates a configuration that uses multiple VMs without an agent to run the compiler tests.

TABLE 4–11 Interview Answers (Compiler Running Locally) That Configure Multiple VMs Without an Agent

Question	Answer
Test Execution Mode — Select the test execution mode	**MultiJVM**
Testing Locally — Are you running these tests on the same system used for this interview?	`Yes`

In the test environment illustrated in Figure 4–7, the JavaTest harness, the compilers under test, and the reference VMs that test the compiler output are run on the same computer but in separate processes. This test environment does not use an agent. Processes are started as required depending on the type of compiler test.

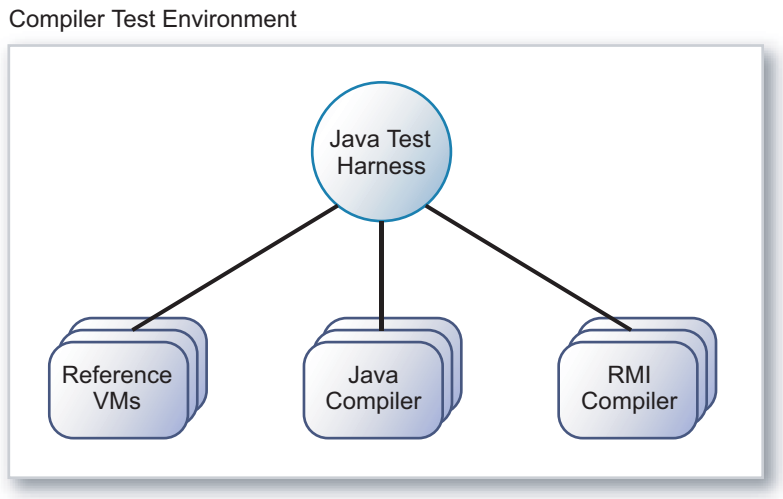


FIGURE 4–7 Compiler Test Environment — Multiple VMs Without an Agent

Note – If the harness displays errors indicating that classes are not found, check that the classpaths were set correctly when configuring the classpath for each of the VMs.

Multiple VMs With an Agent

In the interview, when you provide the answers from Table 4–12, the Configuration Editor creates a configuration that uses multiple VMs with an agent to run the compiler tests.

TABLE 4–12 Interview Answers (Compiler Running Remotely) That Configure Multiple VMs With an Agent

Question	Answer
Test Execution Mode — Select the test execution mode	**MultiJVM**
Testing Locally — Are you running these tests on the same system used for this interview?	No
Test Platform — Which operating system is used on the test platform?	Any choice

In the test environment illustrated in Figure 4–8, the JavaTest harness and the reference VMs that test the compiler output run on the same computer. The agent, the Java compiler and the RMI compiler under test run on another system (the system under test) in separate processes. Processes are started as required depending on the type of compiler test. See Table 4–10 for a description of the different types of compiler tests. See "4.4.2.4 Using Agents" on page 102 for a description of the types of agents that you can use.

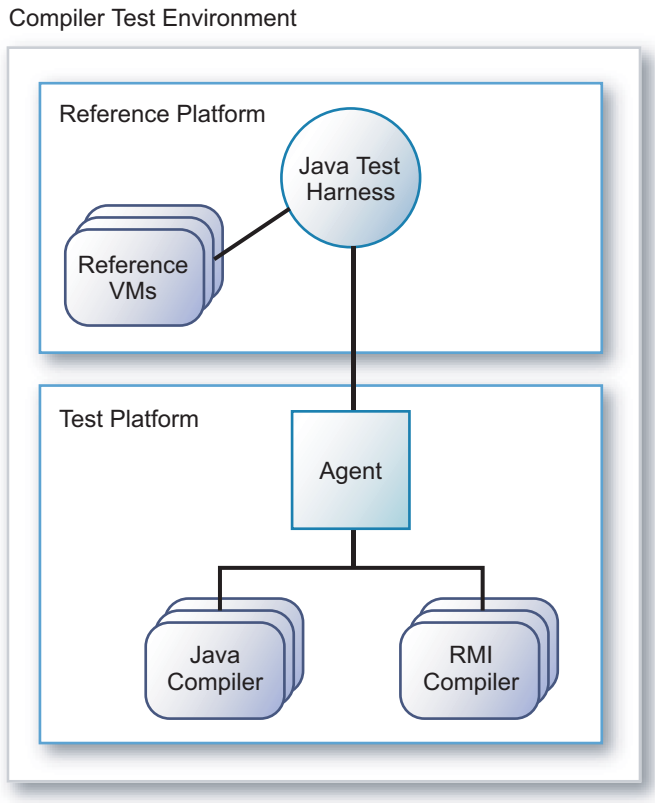


FIGURE 4–8 Compiler Test Environment — Multiple VMs With an Agent

Note – If the harness displays errors indicating that classes are not found, check that the classpaths were set correctly when configuring the classpath for each of the VMs.

Multiple VMs Group Without an Agent

TABLE 4–13 Interview answers that configure Multiple VMs group without an agent

Question	Answer
Test Execution Mode — Select the test execution mode	**MultiJVM group**
Testing Locally — Are you running these tests on the same system used for this interview?	`Yes`

MultiJVM group mode is intended to reduce the overhead of starting VMs by sharing Java complier and Sun reference runtime VM instances among groups of test executions. In this mode the harness starts an internal agent VM on the system that JavaTest is run on and passes a group of the test execution to this agent. Once the internal agent completes its test group run, it closes. The new internal agent VM is started for the next group of tests. The JavaTest harness starts socket server in particular port to communicate with internal agents running on the same system. By default, the port number is chosen from the list of available ports but you can override this parameter.

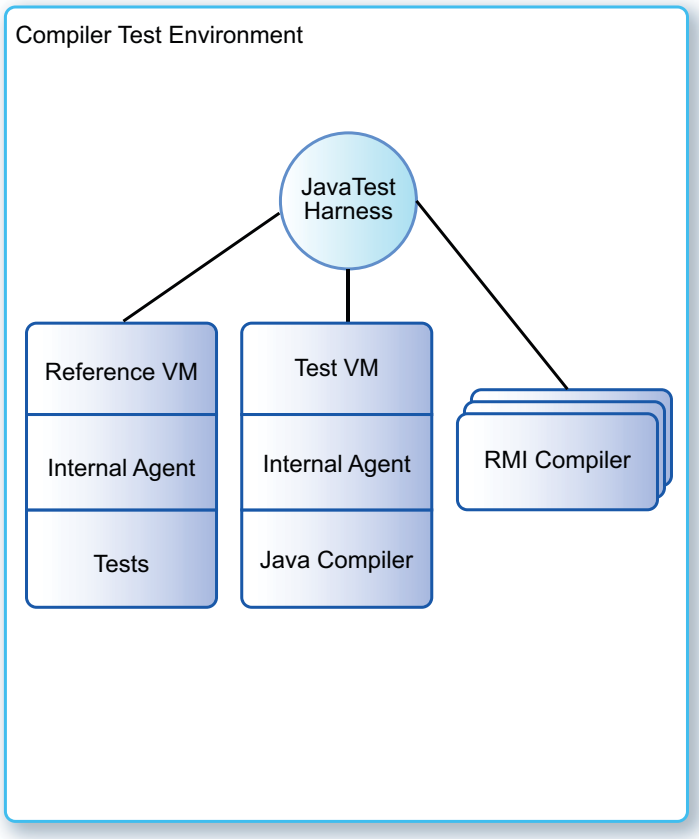


FIGURE 4–9 Compiler Test Environment — Multiple VMs Group Without an Agent

Note – If you have a firewall installed on your system, make sure it allows incoming connections from the local host.

The internal agent waits for a test execution request from the JavaTest harness. If no test request received during a timeout, the agent VM is automatically closed.

Note – The MultiJVM group is not applied to RMI compiler. It is applied only to Java compiler and Sun reference runtime.

The **MultiJVM group** mode expects Java compiler under test to be configured to run via the Java Compiler API interface. The configuration of the **MultiJVM group** mode is the same as the configuration of the **MultiJVM** mode except for several parameters listed in Table 4–14 that control the **MultiJVM group** mode execution.

Note – These parameters are not visible in the JCK configuration interview and can only be changed by setting corresponding test environments variables via the JavaTest harness command line interface. For example:

```
javatest -jar lib/javatest.jar
-Ejck.env.compiler.testCompile.groupMode.serverPort=8080
```

This variable sets TCP port used for listening for incoming connections from internal agents.

See Additional Setup Commands chapter in the JavaTest Harness Command-Line Interface User's Guide.

TABLE 4–14 Environment variables that control execution of the MutliJVM group mode

Variable	Description
jck.env.compiler.testCompile.groupMode.groupSize	Controls a number of tests executed as a group in the same agent VM instance started by the JavaTest harness. Setting groupSize value to *>1* turns the group mode on. Setting groupSize to *1* turns the group mode off. Setting groupSize value to *0* removes limitation on the number of tests executed by one VM (Default is 100). Default group size provides optimal balance between performance and stability of JCK runs.

TABLE 4–14 Environment variables that control execution of the MutliJVM group mode *(Continued)*

jck.env.compiler.testCompile.groupMode.serverPort	A TCP port used for listening for incoming connections from internal agents. Default value is 0, which means that JVM will automatically choose a free port on the test system. You can specify a particular port number to avoid firewall issues.
jck.env.compiler.testCompile.groupMode.timeout	Timeout (in seconds) between test executions during which the agent is active. Default is 60 seconds. Increase this value if necessary.
jck.env.compiler.testCompile.groupMode.agentArgs	Additional agent command line arguments passed to internal agents while starting. For example, you can specify "-trace" option to enable diagnostic output from the agent.
jck.env.compiler.testCompile.groupMode.closeIfFail	A policy used to close agents after test execution if a test status is fail. Default is false, i.e. unset.
jck.env.compiler.testCompile.groupMode.keepOnError	A policy used to close agents after test execution if a test status is error. Default is false, i.e. unset.
jck.env.compiler.testCompile.groupMode.debug	Setting this variable enables additional logging. By default debug is off.

TABLE 4–15 Environment variables that control execution of the MutliJVM group mode for the Sun's reference runtime

Variable	Description
jck.env.compiler.compRefExecute.groupMode.groupSize	Controls a number of tests executed as a group in the same agent VM instance started by the JavaTest harness. Setting groupSize value to *>1* turns the group mode on. Setting groupSize to *1* turns the group mode off. Setting groupSize value to *0* removes limitation on the number of tests executed by one VM (Default is 100). Default group size provides optimal balance between performance and stability of JCK runs.
jck.env.compiler.compRefExecute.groupMode.serverPort	A TCP port used for listening for incoming connections from internal agents. Default value is 0, which means that JVM will automatically choose a free port on the test system. You can specify a particular port number to avoid firewall issues.

TABLE 4–15 Environment variables that control execution of the MutliJVM group mode for the Sun's reference runtime *(Continued)*

jck.env.compiler.compRefExecute.groupMode.timeout	Timeout (in seconds) between test executions during which the agent is active. Default is 60 seconds. Increase this value if necessary.
jck.env.compiler.compRefExecute.groupMode.agentArgs	Additional agent command line arguments passed to internal agents while starting. For example, you can specify "-trace" option to enable diagnostic output from the agent.
jck.env.compiler.compRefExecute.groupMode.closeIfFail	A policy used to close agents after test execution if a test status is fail. Default is false, i.e. unset.
jck.env.compiler.compRefExecute.groupMode.keepOnError	A policy used to close agents after test execution if a test status is error. Default is false, i.e. unset.
jck.env.compiler.compRefExecute.groupMode.debug	Setting this variable enables additional logging. By default debug is off.

Note – When you select the **MutliJVM group** mode, JCK does its best to optimize execution by grouping tests and executing them in the same VM instance. However, in case tests cannot be grouped (for example, tests marked by the keyword "only_once"), the tests will be executed individually.

Multiple VMs Group With an Agent

TABLE 4–16 Interview answers that configure Multiple VMs group with an agent

Question	Answer
Test Execution Mode — Select the test execution mode	**MultiJVM group**
Testing Locally — Are you running these tests on the same system used for this interview?	**No**
Test Platform	**Other**

When you answer interview questions as specified in Table 4–16, the JavaTest harness runs on a separate computer in the test environment (using one of the reference VMs) while an agent runs tests in separate VMs system under test. The agent, JCK tests, and the compiler run in separate Java runtime environments on the target system. The agent starts a new internal agent VM on the target system and passes a group of test execution to this agent. However, the reference runtime is started on the system that JavaTest is run.

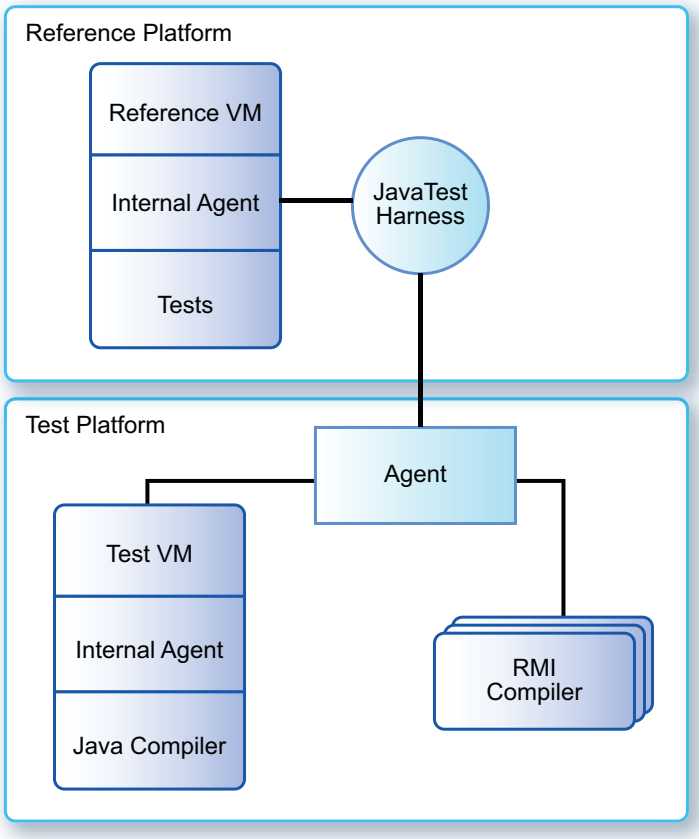


FIGURE 4–10 Compiler Test Environment — Multiple VMs Group With an Agent

All Multi JVM group mode configuration parameters, mentioned in Table 4–14 apply to this execution mode. See "4.4.2.4 Using Agents" on page 102 for a description of the types of agents that you can use.

Single VM With an Agent

In the interview, when you select **SingleJVM** in the following question, the Configuration Editor creates a configuration that uses a single VM with an agent to run the compiler tests:

Test Execution Mode — Select the test execution mode

In the test environment illustrated in Figure 4–11, the JavaTest harness and the reference VM that tests the compiler output are run on the same computer. An agent, the Java compiler, and

the RMI compiler run in the same process on another system (the system under test). This test configuration is used when the test system does not support multiple processes.

The configuration for single VM test commands that use an agent to invoke a compiler is different from the multiple VM commands. To achieve a better performance, the command, executing reference runtime, is the same as in the **Multi JVM group** mode in which the harness starts an internal agent VM on the system that JavaTest is run on and passes a group of the test execution to this agent. You can change this to use **Multi JVM** mode by setting the environment variable `jck.env.compiler.compRefExecute.groupMode.groupSize` to 1. See Table 4–15 *Environment variables that control execution of the MutliJVM group mode for the Sun's reference runtime* for more information.

Because the agent and the compilers run in the same VM the classpath for the agent VM must be correct for the agent and the compilers. See "4.4.2.4 Using Agents" on page 102 for a description of the types of agents that you can use.

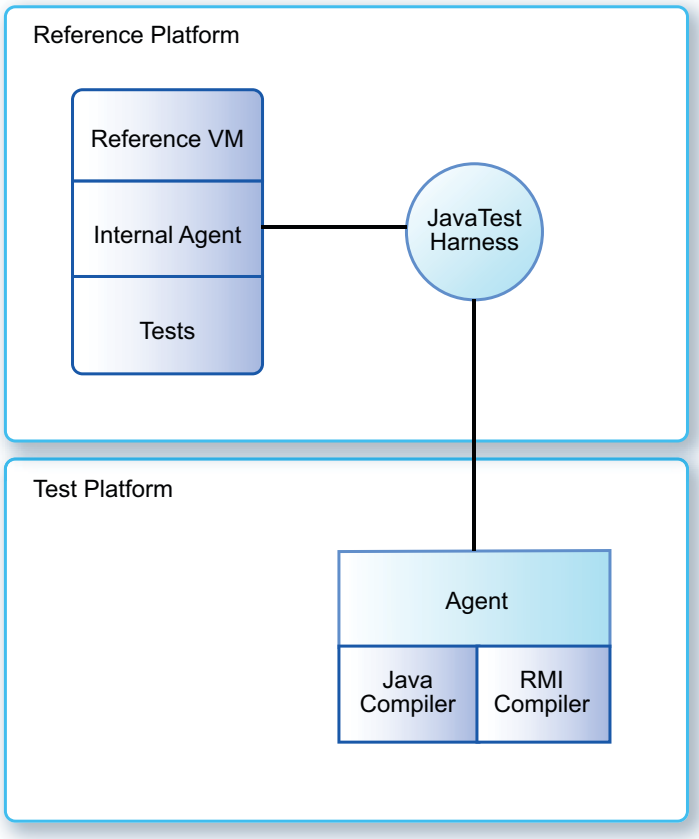


FIGURE 4–11 Compiler Test Environment — Single VM With an Agent

Note – If the harness displays errors indicating that classes are not found, check that the classpaths were set correctly when configuring the class path for each of the VMs.

4.4.2.3 JCK Devtools Test Suite

The interview presents a series of questions and uses their answers to create a configuration for the test environment of the JCK devtools (devtools) test suite. The following sections describe the possible configurations of test environments used to run the tests of the devtools test suite:

- "Multiple VMs Without an Agent" on page 93
- "Multiple VMs With an Agent" on page 94
- "Multiple VMs Group Without an Agent" on page 96
- "Multiple VMs Group With an Agent" on page 99

- "Single VM With an Agent" on page 100

It is important to note the following descriptions of how the devtools tests work:

- The schema compiler compiles test files and executes the resulting class files on a Sun reference runtime implementation.
- The schema generator generates schemas from test files and validates the resulting schemas on the Sun reference runtime implementation.

Both the schema compiler and the schema generator are types of compilers. The schema compiler maps from schemas to Java class files. The schema generator maps from Java class files to schemas.

Note – Devtools compiler is used in this section to refer to both the schema compiler and the schema generator.

To compile and execute devtools compiler tests, you must configure three test commands that invoke the following:

- Schema compiler
- Schema generator
- Sun reference runtime

Table 4–17 describes the different types of devtools compiler tests.

TABLE 4–17 Types of Devtools Compiler Tests

Type of Test	Processes Started
Negative (Schema Compiler)	Only the Schema compiler is started, the compiler is expected to fail and produce an error. A new compiler is started for every test.
Positive (Schema Compiler)	A new Schema compiler is started for every test. Output of the compiler is always checked using a new reference VM.
Positive (Schema Generator)	A new Schema generator is started for every test. Output of the generator is always checked using a new reference VM.

Multiple VMs Without an Agent

In the interview, when you provide the answers from Table 4–11, the Configuration Editor creates a configuration that uses multiple VMs without an agent to run the devtools compiler tests.

TABLE 4–18 Interview Answers (Devtools Compiler Running Locally) That Configure Multiple VMs Without an Agent

Question	Answer
Test Execution Mode — Select the test execution mode	**MultiJVM**
Testing Locally — Are you running these tests on the same system used for this interview?	**Yes**

In the test environment illustrated in Figure 4–12, the JavaTest harness, the devtools compiler under test, and the reference VMs that test the devtools compiler output are run on the same computer but in separate processes. This test environment does not use an agent. Processes are started as required depending on the type of devtools compiler test.

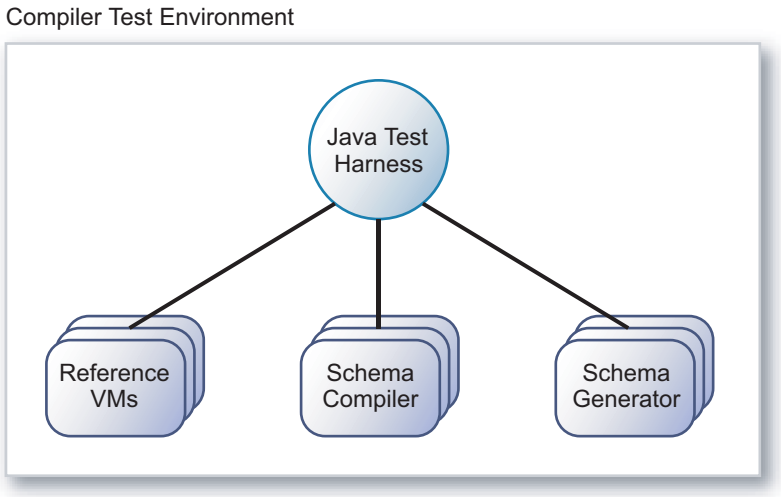


FIGURE 4–12 Devtools Compiler Test Environment — Multiple VMs Without an Agent

---

Note – If the harness displays errors indicating that classes are not found, check that the classpaths were set correctly when configuring the classpath for each of the VMs.

Multiple VMs With an Agent

In the interview, when you provide the answers from Table 4–12, the Configuration Editor creates a configuration that uses multiple VMs with an agent to run the devtools compiler tests.

TABLE 4–19 Interview Answers (Devtools Compiler Running Remotely) That Configure Multiple VMs With an Agent

Question	Answer
Test Execution Mode — Select the test execution mode	**MultiJVM**
Testing Locally — Are you running these tests on the same system used for this interview?	**No**

In the test environment illustrated in Figure 4–13, the JavaTest harness and the reference VMs that test the devtools compiler output run on the same computer. The agent and the devtools compilers run on another system (the system under test) and in separate processes. Processes are started as required depending on the type of compiler test. See Table 4–17 for a description of the types of compiler tests. See "4.4.2.4 Using Agents" on page 102 for a description of the types of agents that you can use.

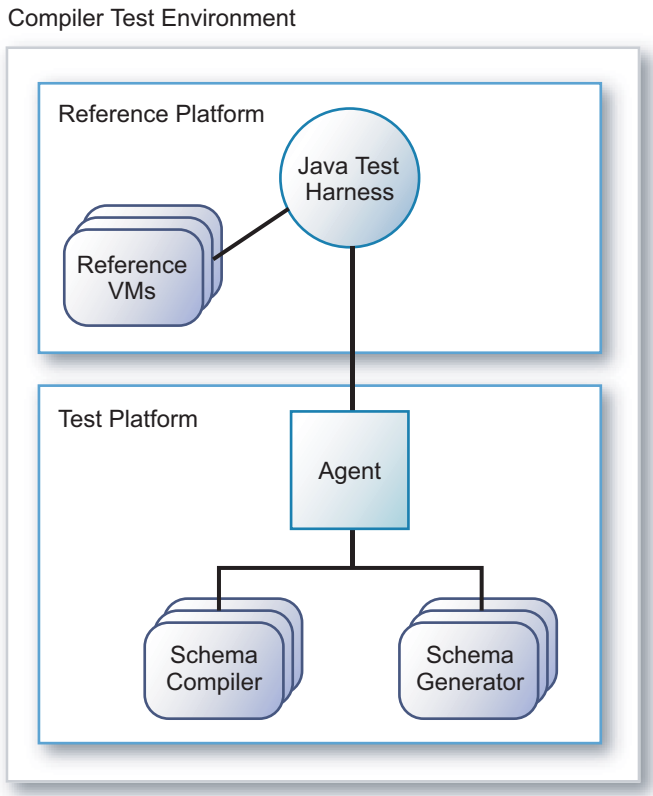


FIGURE 4–13 Devtools Compiler Test Environment — Multiple VMs With an Agent

Note – If the harness displays errors indicating that classes are not found, check that the classpaths were set correctly when configuring the classpath for each of the VMs.

Multiple VMs Group Without an Agent

TABLE 4–20 Interview answers (Devtools Compiler Running Locally) that configure Multiple VMs group without an agent

Question	Answer
Test Execution Mode — Select the test execution mode	**MultiJVM group**
Testing Locally — Are you running these tests on the same system used for this interview?	`Yes`

MultiJVM group mode is intended to reduce the overhead of starting VMs by sharing Sun reference complier and Sun reference runtime VM instances among groups of devtools test executions. In this mode the harness starts an internal agent VM on the system that JavaTest is run on and passes a group of the test execution to this agent. Once the internal agent completes its test group run, it closes. The new internal agent VM is started for the next group of tests. The JavaTest harness starts socket server in particular port to communicate with internal agents running on the same system. By default, the port number is chosen from the list of available ports but you can override this parameter.

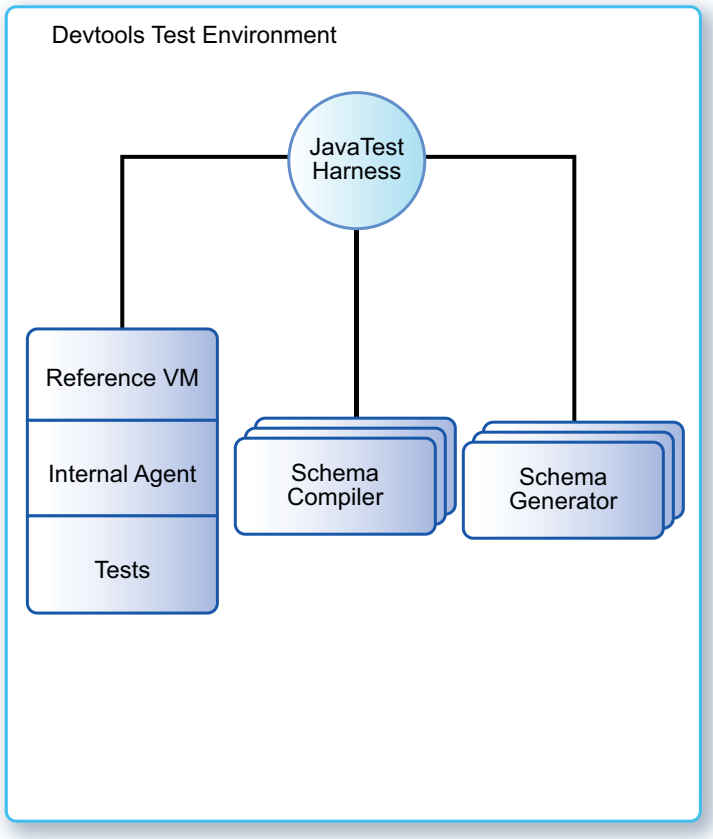


FIGURE 4–14 Devtools Compiler Test Environment — Multiple VMs Group Without an Agent

---

Note – If you have a firewall installed on your system, make sure it allows incoming connections from the local host.

The internal agent waits for a test execution request from the JavaTest harness. If no test request received during a timeout, the agent VM is automatically closed.

Note – The MultiJVM group is not applied to Devtools compilers. It is applied only to Sun reference runtime and Sun reference Java compiler.

Note – In this mode, the interview question about the path to Sun reference java compiler will not be asked. The reference compiler will be automatically configured to be run via Java Compiler API interface using Sun reference runtime.

The configuration of the **MultiJVM group** mode is the same as the configuration of the **MultiJVM** mode except for several parameters listed in Table 4–14 that control the **MultiJVM group** mode execution.

Note – These parameters are not visible in the JCK configuration interview and can only be changed by setting corresponding test environments variables via the JavaTest harness command line interface. For example:

`javatest -jar lib/javatest.jar -Ejck.env.devtools.testExecute.groupMode.groupSize=400` This variable sets the number of tests grouped in one VM instance.

See Additional Setup Commands chapter in the JavaTest Harness Command-Line Interface User's Guide.

TABLE 4–21 Environment variables that control execution of the MultiJVM group mode

Variable	Description
jck.env.devtools.testExecute.groupMode.groupSize	Controls a number of tests executed as a group in the same agent VM instance started by the JavaTest harness. Setting groupSize value to *>1* turns the group mode on. Setting groupSize to *1* turns the group mode off. Setting groupSize value to *0* removes limitation on the number of tests executed by one VM (Default is 100). Default group size provides optimal balance between performance and stability of JCK runs.
jck.env.devtools.testExecute.groupMode.serverPort	A TCP port used for listening for incoming connections from internal agents. Default value is 0, which means that JVM will automatically choose a free port on the test system. You can specify a particular port number to avoid firewall issues.
jck.env.devtools.testExecute.groupMode.timeout	Timeout (in seconds) between test executions during which an agent is active. Default is 60 seconds. Increase this value if necessary.

TABLE 4–21 Environment variables that control execution of the MultiJVM group mode *(Continued)*

jck.env.devtools.testExecute.groupMode.agentArgs	Additional agent command line arguments passed to internal agents while starting. For example, you can specify "-trace" option to enable diagnostic output from the agent.
jck.env.devtools.testExecute.groupMode.closeIfFail	A policy used to close agents after test execution if a test status is fail. Default is false, i.e. unset.
jck.env.devtools.testExecute.groupMode.keepOnError	A policy used to close agents after test execution if a test status is error. Default is false, i.e. unset.
jck.env.devtools.testExecute.groupMode.debug	Setting this variable enables additional logging. By default debug is off.

Note – When you select the **MutliJVM group** mode, JCK does its best to optimize execution by grouping tests and executing them in the same VM instance. However, in case tests cannot be grouped (for example, tests marked by the keyword "only_once"), the tests will be executed individually.

Multiple VMs Group With an Agent

TABLE 4–22 Interview answers (Devtools Compiler Running Remotely) that configure Multiple VMs group with an agent

Question	Answer
Test Execution Mode — Select the test execution mode	**MultiJVM group**
Testing Locally — Are you running these tests on the same system used for this interview?	`No`

When you answer interview questions as specified in Table 4–22, the JavaTest harness runs on a separate computer in the test environment (using one of the reference VMs) while an agent runs tests in separate VMs system under test. The agent, JCK tests, and the devtools compiler run in separate Java runtime environments on the target system. The agent starts a new internal agent VM on the target system and passes a group of test execution to this agent. However, the reference runtime is started on the system that JavaTest is run.

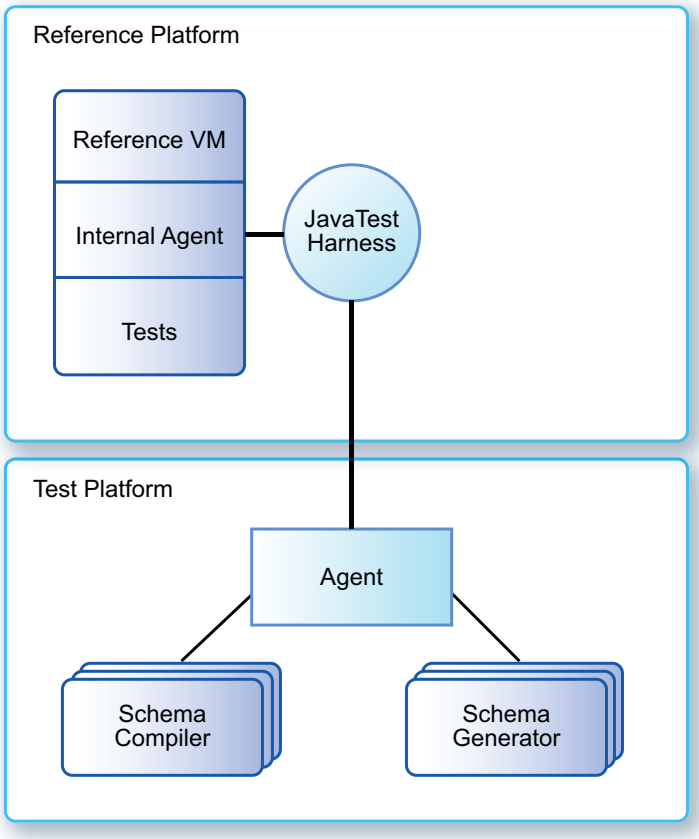


FIGURE 4–15 Devtools Compiler Test Environment — Multiple VMs Group With an Agent

All **Multi JVM group** mode configuration parameters, mentioned in Table 4–21 apply to this execution mode. See "4.4.2.4 Using Agents" on page 102 for a description of the types of agents that you can use.

Single VM With an Agent

In the interview, when you select **SingleJVM** in the following question, the Configuration Editor creates a configuration that uses a single VM with an agent to run the devtools compiler tests:

Test Execution Mode — Select the test execution mode

Note – In this test environment, the harness uses an agent when running tests locally or remotely.

In the test environment illustrated in Figure 4–16, the JavaTest harness and the reference VM that tests the compiler output are run on the same computer while the agent and the compilers under test run on another system (the system under test) in the same VM. This test configuration is used when the test system does not support multiple processes.

The configuration for single VM test commands that use an agent to invoke a compiler is different from the multiple VM commands. To achieve better performance, the command, executing reference runtime and reference compiler, is the same as in the **Multi JVM group** mode, in which the harness starts an internal agent VM on the system that JavaTest is run on and passes a group of the test execution to this agent. You can change this to use **Multi JVM** mode by setting the environment variable `jck.env.devtools.testExecute.groupMode` to 1. See chapter "Multiple VMs Group Without an Agent" on page 96 for more information about the group mode.

Because the agent and the compilers run in the same VM, the classpath for the agent VM must be correct for both the agent and the compilers. See "4.4.2.4 Using Agents" on page 102 for a description of the types of agents that you can use.

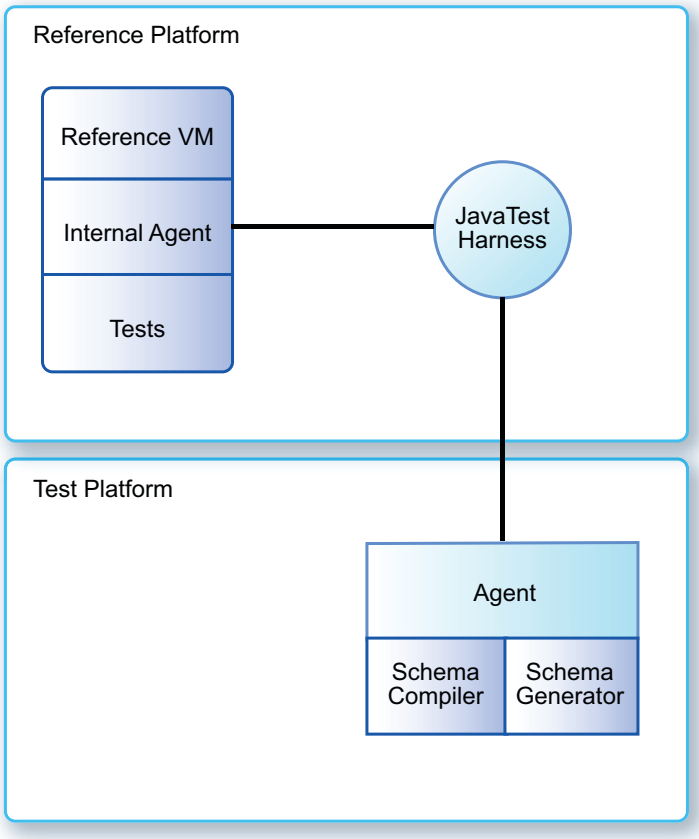


FIGURE 4–16 Devtools Compiler Test Environment — Single VM With an Agent

Note – If the harness displays errors indicating that classes are not found, check that the classpaths were set correctly when configuring the classpath for each of the VMs.

4.4.2.4 Using Agents

When you configure your test environment, you specify whether you use an *active* or *passive* agent to run the tests. Active agents initiate the connection with the JavaTest harness, as opposed to passive agents that require the JavaTest harness to initiate the connection.

If you are using the JavaTest harness agent provided by the JavaTest harness, see the *JavaTest Agent User's Guide* or the JavaTest harness online help for detailed instructions about configuring and using JavaTest harness agents to run tests.

If you are using another agent, see the documentation provided with that agent for detailed instructions about configuring and using the agent to run tests.

4.4.3 Test-Specific Questions

The second section of each interview contains sub interviews that collect configuration information about the tests to be run. Each interview uses the More Info panel of the configuration editor to provide specific test configuration information based on your answers to previous interview questions.

For additional information about special setup and configuration requirements for running specific tests in the test suite, see Chapter 5.

4.4.4 Test Run Questions

The last section of each interview specifies how to run the tests. For example, you can run tests concurrently, set the time factor, and include or exclude tests from the test run by using keywords and using exclude lists.

See the *JavaTest Harness User's Guide: Graphical User Interface* included with this release or the JavaTest harness online help for detailed instructions about using the configuration editor window.

Tip – In Simple mode, the**Test Run Questions** section of the configuration is hidden. You can access these values via the JavaTest GUI using the Quick Set Mode interview.

4.4.4.1 Exclude Lists

In the command line or in the GUI configuration editor window, you can specify the exclude lists used when running tests. You can choose the exclude list provided by the JCK 6b test suite or you can run the tests without using an exclude list. If your system has Internet access, you can also configure the JavaTest harness to automatically check for and download updated JCK 6b exclude lists before running tests. See the *JavaTest Harness User's Guide: Graphical User Interface*, the *JavaTest Harness User's Guide: Command-Line Interface*, or the JavaTest harness online help for a detailed description of specifying one or more exclude lists when running tests.

4.4.4.2 Keywords

Each test can contain one or more keywords defined by the JCK test suite that can be used to restrict the set of tests that are run. Keywords can also be associated with different groups of tests within the test suite. In the command line or in the GUI configuration editor window, you can specify the keywords used to filter the tests that are run. See the *JavaTest Harness User's*

Guide: Graphical User Interface, the *JavaTest Harness User's Guide: Command-Line Interface*, or the JavaTest harness online help for a detailed description specifying keywords used when running tests. See "A.1.3 Keywords" on page 231 for more information about keywords.

4.4.5 Configuration Question Log

When the Configuration Editor saves a configuration, it generates a Configuration Question Log that lists each question asked, the tag name of the question, and the corresponding answer.

The Configuration Question Log can be used during troubleshooting to quickly review the contents of the current configuration interview.

See the *JavaTest Harness User's Guide: Graphical User Interface* or the JavaTest harness online help for a detailed description of using the Configuration Question Log.

4.4.6 Configuration Checklist

When the Configuration Editor saves a configuration, it generates a Configuration Checklist from each interview. Some tests require additional setup steps (such as starting additional programs) prior to running them. You can use the Configuration Checklist to verify that all required preparations are complete before running tests.

Note – Because the Configuration Editor generates the checklist from the completed interview, it only lists those setup steps actually required by that configuration to run tests.

See the *JavaTest Harness User's Guide: Graphical User Interface* or the JavaTest harness online help for a detailed description of using the Configuration Checklist.

4.5 Special Setup Instructions

Special setup instructions are described on a test-by-test basis in Chapter 5 as well as in the Configuration Checklist. See "4.4.6 Configuration Checklist" on page 104 for information about the Configuration Checklist. Table 4–23 lists the tests that require additional setup and the page containing the setup instructions.

TABLE 4–23 Special Test Setup Instructions

Test	Section and Page Number
Annotation Processing Tests	"5.2.1 Setup" on page 112
AWT and Swing Tests	"5.3.1 Setup" on page 114
Compiler Tests	"5.4.1 Setup" on page 118
CORBA Tests	"5.5.1 Setup" on page 120
Distributed Tests	"5.6.1 Setup" on page 123
Extra-Attribute Tests	"5.7.1 Setup" on page 126
ImageIO Tests	"5.9.1 Setup" on page 133
Interactive Tests	"5.10.1 Setup" on page 134
JAX-WS Mapping Tests	"5.11.1 Setup" on page 136
JDBC Technology RowSet Tests	"5.12.1 Setup" on page 140
JDWP Tests	"5.13.1 Setup" on page 142
JMX Advanced Dynamic Loading Tests	"5.14.1 Setup" on page 146
JMX Remote API Tests	"5.15.1 Setup" on page 154
JNI Implementation Tests	"5.16.1 Setup" on page 157
JVM TI Tests	"5.17.1 Setup" on page 164
Java Generic Security Service API Tests	"5.19.1 Setup" on page 176
Java Programming Language Instrumentation Services Tests	"5.20.1 Setup" on page 181
Java Platform Scripting API Tests	"5.21.1 Setup" on page 185
Java RMI Tests	"5.23.1 Setup" on page 188
Network Tests	"5.25.1 Setup" on page 193
Printing Tests	"5.28.1 Setup" on page 201
Schema Compiler	"5.29.1 Setup" on page 202
Schema Generator	"5.30.1 Setup" on page 205
Security Tests	"5.31.1 Setup" on page 208
Static Initialization Tests	"5.33.1 Setup" on page 213

4.6 Special Test Execution Requirements

The following tests require multiple runs using different configurations:

VM Tests — Test specific information required to run the VM tests in all required configurations is provided in the VM tests Execution section of Chapter 5.

4.7 JCK-devtools Test Suite Execution Requirements

This section includes special considerations, necessary steps, and preliminary tasks required to run JCK-devtools test suite tests.

- Agent classpath should contain the path to the jtjck.jar file to run Schema Compiler, Schema Generator, and JAX-WS tests.
- When you run schema compilation through the JavaTest Agent, your schema compiler will be executed in the Agent's JVM. Your implementation should provide a custom class, which implements the following interface:

```
package com.sun.jck.lib;

public interface SchemaCompilerTool {

/**

* @param xsdFiles array of strings containing schema files

* @param packageName the target package

* @param outDir output directory where java file(s) will be generated

* @param out output stream for logging

* @param err error stream for logging

* @return 0 if java file(s) generated successfully

*/

int compile(String[] xsdFiles, String packageName, File outDir, PrintStream out,
PrintStream err);

}
```

  Schema compilation should be accomplished during invocation of compile method. The JCK devtools test suite includes the following class that is fully compatible with Sun's reference implementation and that can be used for schema compilation: `com.sun.jck.lib.SchemaCompiler`. The realization of the method `compile(String[] xsdFiles, String packageName, File outDir, PrintStream out, PrintStream err)` compiles one or more schemas into java sources.

- When you run schema generation through JavaTest Agent, your generator will be executed in the Agent's JVM. Your implementation should provide a custom class, which implements the following interface:

  ```
  package com.sun.jck.lib;

  public interface SchemaGenTool {

  int generate(String[] args, PrintStream out, PrintStream err);

  }
  ```

  The invocation of the `generate` method should start schema generation.

  The JCK devtools test suite includes the following class that is fully compatible with Sun's reference implementation and that can be used for schema generation: `com.sun.jck.lib.SchemaGen`. The realization of the method `generate(String[] args, java.io.PrintStream out, java.io.PrintStream err)` generates xml schema from one or more java files.

- JCK provides commands to invoke your schema compiler. For example, for the Sun JAXB implementation on Solaris, Sun provides a script `solaris/bin/xjc.sh` that compiles schemas into class files, the script can be found at the following location:

  `solaris/bin/xjc.sh`

  For the Sun JAXB implementation on Windows, Sun provides a script win32\bin\xjc.bat, which compiles schemas into class files, the command can be found at the following location:

  `win32\bin\xjc.bat`

  The xjc script supports the following mandatory command-line options:

  Usage: xjc –cp <arg> –p <pkg> –d <dir> <schema file>

  Options:

  - –cp <arg> — specifies class search path
  - –p <pkg> — specifies the target package
  - –d <dir> — target directory to store generated files

- JCK provides commands to invoke your schema generator. For example, for the Sun JAXB implementation on Solaris, Sun provides a script solaris/bin/schemagen.sh that generates schemas from java files, the script can be found at the following location:

  `solaris/bin/schemagen.sh`

  For the Sun JAXB implementation on Windows, Sun provides a script win32\bin\schemagen.bat that compiles schemas into class files, the script can be found at the following location:

  `win32\bin\schemagen.bat`

  The schemagen script supports the following mandatory command-line options:

Usage: schemagen –cp <arg> –d <dir> <java files>

Options:

- –cp <arg> — specifies class search path
- –d <dir> — target directory to store generated files

◆ ◆ ◆ CHAPTER 5

Test-Specific Information

This chapter provides special setup instructions, configuration information, and special execution instructions required to run the following JCK tests:

The configuration is generated from each interview by the Configuration Editor. See Chapter 4 for additional information about creating configurations for test runs.

Note – The configuration tab value names and descriptions provided in the sections of this chapter are used with the JavaTest harness Configuration tab and the Test Environment dialog box during troubleshooting.

You can also set specific configuration values in the command line for a test run. If you choose to do so, see the *JavaTest Harness User's Guide: Command-Line Interface*, or the JavaTest harness online help for a detailed description of setting configuration values in the command line. You must use one of the following sources to identify the question-tag name to use in the command line:

- The Question Log file (`questionLog`*nnnn*`.html`) generated by the Configuration Editor in *workdirectory*`/jtData`
- The `config.html` report generated from the JavaTest harness GUI
- The Question Tags displayed in the Configuration Editor

See the *JavaTest Harness User's Guide: Graphical User Interface* or the JavaTest harness online help for a detailed description of displaying the Question Tags in the Configuration Editor and generating reports containing the `config.html` file.

Note – The `questionLog`*nnnn*`.html` and `config.html` files can be viewed in any web browser.

Configuration values that you set in the command line are not saved in the configuration file. Use the Configuration Editor or the command line `Edit JTI` utility to modify the configuration file. See the *JavaTest Harness User's Guide: Graphical User Interface*, the *JavaTest Harness User's Guide: Command-Line Interface*, or the JavaTest harness online help for a detailed description of using the Configuration Editor or the `Edit JTI` utility to modify a configuration.

5.1 How This Chapter Is Organized

This chapter is organized alphabetically according to the major test categories of each interview. Each test category has four sections (see Table 5–1) that describe information specific to the different types of JCK tests.

TABLE 5–1 Test Information Sections and Descriptions

Information Section	Description
Setup	Special steps (if any) required to setup specific tests prior to running them.
Configuration	Configuration information (if any) required to run specific tests.
Execution	Special instructions (if any) required to execute specific tests.
Configuration Tab Name-Value Pairs	Configuration tab name-value pairs are provided in this chapter for you to use with the JavaTest harness Configuration tab or Test Environment dialog box when troubleshooting problems running a test. When viewing Configuration tab name-value pairs in this chapter, note that questions in the interview, properties in the configuration file, and the actual parameters of the test could be different. For example, the JavaTest harness uses the TestScript plugin to calculate some test parameters from configuration file values. If you set a configuration value in the command line, you must use the Question Log, the `config.html` report, or the Question Tags in the Configuration Editor to identify the question-tag name used in the command line for that configuration. See Appendix A for a detailed list of entry names and values used in the test description table.

The majority of test-specific information pertains only to the runtime tests. Java platform Remote Method Invocation (Java RMI) and floating-point tests are the only compiler tests that require any special action.

5.2 Annotation Processing Tests

Annotation processing testing is a three-step process in which the input `.java` source file is processed using the Java compiler and annotation processing options, the annotation processor writes the test status into a file in the test work directory, and the harness checks the test execution status file created by the processor. Table 5–2 provides the location of the annotation processing tests and identifies the areas that are tested.

TABLE 5–2 Annotation Processing Test Information

Test Information	Description or Value
APIs tested	`javax.annotation.processing` `javax.lang.model` and subpackages
Test URL	`api/javax_annotation/processing` `api/javax_lang/model`

5.2.1 Setup

When rerunning the annotation processing tests, verify that the work area does not contain class files created during a previous test run.

5.2.2 Configuration

Each annotation processing test consists of an annotation processor and an input `.java` source file. The annotation processor contains the test code that checks the specification assertion. The input `.java` source file, when correctly processed by the compiler, produces structured output expected by the test.

Note – The configuration steps for annotation processing tests described in this chapter are only needed for the command line compiler. The JSR 199 compiler uses the standard API and does not require the additional configuration.

To create a configuration for running the annotation processing tests, you must specify the following templates in the interview:

- **Processor template** — Passes the processor class name to a compiler. Each test has its own processor class. The class name is stored in the `apClass` test description field.
- **Processor specific options template** — Passes the processor options. The following options are two JCK standard options that are defined by harness and passed to test:
  - `jckTestWorkDir` contains the test work directory. The value passed in the `jckTestWorkDir` option is used by processor (test) to write the file that contains the test execution status. After the test is complete, the harness reads the status file from that directory and detects whether the test has passed or failed results. The tests write their execution status into files located in the specified directory. The JavaTest harness uses these files to detect whether the test has passed or failed results.
  - `jckApArgs` contains the value specified in the test description `apArgs` field. The `apArgs` field is an additional way to pass the options to a test by analogy with `executeArgs`. Not every test has `apArgs` field.

Note – Each test has its own test work directory and processor arguments.

- **Source output directory** — Stores the files generated by the processor.

The harness uses the templates and values from the configuration to generate the compiler command for annotation processing tests. The compiler command generated by the harness passes the annotation processor, source output directory, and processor-specific option to the Java compiler.

Table 5–3 provides the names and descriptions of the templates required to create a configuration for running the annotation processing tests.

TABLE 5–3 Annotation Processing Test Configuration Templates

Template Name	Template Description
Annotation Processor Option	A template for the processor option. In the option template, the pound sign (#) indicates where the processor class name is specified. Example: `-processor #`
Source Output Directory	A template for the source output directory option. In the option template, the pound sign (#) indicates where the source output directory name is specified. Example: `-s #`
Processor Specific Options	A template for passing processor-specific options to the annotation processing tool. It is used to pass key-value pairs to the Annotation Processing test. Example: `-A#key=#value`

Following is an example of the annotation processing command line generated by the harness for the standard JDK software on the Solaris operating system:

```
... -processor $apClass -AjckTestWorkDir=$testWorkDir \
-AjckApArgs=$apArgs -s $testWorkDir ... $testSource
```

In the example, the following options and values are used:

- `$apClass` — Annotation processor class
- `jckTestWorkDir` — Special option name for passing the test work directory

- `$testWorkDir` — Work directory for a test
- `jckApArgs` — Special option name for passing the additional arguments to a test
- `$testSource` — `.java` source file to be processed by the processor

5.2.3 Execution

No special requirements.

5.2.4 Configuration Tab Name-Values Pairs

The user must enter templates as described in Table 5–3. The harness uses the templates and values from the configuration to generate the name-value pairs displayed in the GUI.

5.3 AWT and Swing Tests

Table 5–4 provides the location of the AWT and Swing tests and identifies the areas that are tested.

TABLE 5–4 AWT and Swing Test Information

Test Information	Description or Value
APIs tested	`java.awt` and subpackages `javax.swing` and subpackages
Test URLs	`api/java_awt` `api/javax_swing`

5.3.1 Setup

The AWT and Swing tests contain batch and interactive tests. Because interactive tests require keyboard and mouse input, you might decide to run them separately and not batched with the rest of the test suite. All interactive tests contain the `interactive` keyword. In addition to the `interactive` keyword, some AWT interactive tests contain the `robot` keyword. This enables you to specify keywords used to filter or include these tests in a test run. In addition, if your platform supports it, many of the tests that might require human interaction can also be run without interaction by using the `java.awt.Robot` class. Tests that use the robot can be selected or deselected by specifying the `robot` keyword in the configuration. For more information about using keywords in the configuration, see Chapter 4.

Running the interactive tests separately reduces the time required to find the tests that use the `interactive` or `robot` keywords. To run the interactive tests separately, specify in the configuration that only AWT and Swing tests are run. See Chapter 4 for detailed information.

The AWT interactive tests are located at `api/java_awt/interactive/index.html`.

The Swing interactive tests are located at `api/javax_swing/interactive/index.html`. For more details, see "5.10 Interactive Tests" on page 134.

5.3.2 Configuration

When testing on X11 window systems, the `DISPLAY` and `HOME` variables in Table 5–5 must be set in the test environment.

5.3.3 Execution

Some AWT and Swing tests require user interaction. Interactive tests have been developed using two kinds of user interfaces: a Yes-No interface and a Done interface. Tests that use the Yes-No interface request the user to perform a visual verification and keyboard-mouse actions, and to click Yes or No to declare whether the test passed or failed. If the user clicks No in the case of a failed test, the user is prompted to enter text that describes the failure. That text is included in the test's results file (`.jtr`). Tests that use the Done interface request the user to perform keyboard-mouse actions and click Done.

Interactive tests time out if the user does not click Yes-No or Done within a certain amount of time. Timeout values for these tests vary. The timeout value for a test is based on the complexity of the test. A test must be rerun if it times out, perhaps with a different timeout value.

Note – You can use the Pause-Resume button in each interactive test frame to pause the test timer and then work with the test as long as you want.

Some automated interactive tests use the `java.awt.Robot` class to generate key presses. On some platforms, the delay between key presses is a user-configurable property and the behavior of these automated interactive tests might be affected by the delay between key presses. The user might need to adjust the delay to make the tests execute correctly.

Interactive AWT tests set the size of the test frame to the size of the screen if the screen size is smaller than the size of the test frame.

Some automated interactive tests fail if they are run on low-resolution screens when using the JavaTest harness command-line interface. All components must be visible in the test frame for the robot to interact with them.

Note – Interactive tests that test or use the robot can fail if all components in the test frame are not visible.

If you run an automated test in this situation, the test fails with the following message:

```
Cannot run automated tests on small screen.
Please set robotAvailable to false and rerun
the tests.
```

If this occurs, set the `robotAvailable` configuration value to false and change your answer to the AWT Robot interview question to No. For details about running interactive tests, see "5.10 Interactive Tests" on page 134.

5.3.4 Configuration Tab Name-Value Pairs

The Configuration tab name-value pairs in Table 5–5 are provided for use with the JavaTest harness test view and Test Environment dialog box when troubleshooting problems with running tests. You can view the configuration values that were used to run a specific test by choosing the Configuration tab in the Test Manager.

For a description of the test view tabs and Test Environment dialog box, see the *JavaTest Harness User's Guide: Graphical User Interface* included with this release or the JavaTest harness online help.

If you intend to set a configuration value in the command line to run tests, see Chapter 4 for a description of how to obtain the required question-tag names for your configuration.

Table 5–5 lists the name-values pairs displayed in the Configuration tab for the AWT and Swing tests.

TABLE 5–5 AWT and Swing Test Configuration Tab Name-Value Pairs

Name	Value Description
`DISPLAY`	Users on X11-based systems must define the `$DISPLAY` value explicitly in the configuration. If the value is not set in the configuration, you can set it as a system property by using the `-D` switch when starting the JavaTest harness. For example:`java -DDISPLAY=$DISPLAY \ com.sun.javatest.tool.Main` This value is automatically set for you by the script `jck/solaris/bin/javatest`.
`HOME`	Users on X11-based systems must define the `$HOME` value explicitly in the configuration. If the value is not set in the configuration, you can set it as a system property by using the `-D` switch when starting the JavaTest harness. For example:`java -DHOME=$HOME \ com.sun.javatest.tool.Main` This value is automatically set for you by the script `jck/solaris/bin/javatest`.
`platform.isHeadless`	Boolean value that indicates whether the system being tested is headless. A headless system is one that does not have a display device, a keyboard, or mouse. If the system is headless, this value must be `true`. Otherwise, it must be `false`.
`platform.robotAvailable`	Boolean value that indicates whether the system being tested supports low-level input control (provided by `java.awt.Robot` classes). If the system supports the `java.awt.Robot` classes, the value must be `true`. Otherwise, it must be `false`.

5.4 Compiler Tests

Table 5–6 provides the location of the compiler tests and identifies the areas that are tested.

TABLE 5–6 Compiler Test Information

Test Information	Description or Value
APIs tested	Not applicable for these tests
Test URL	`lang`

5.4.1 Setup

When rerunning compiler tests, verify that the work area does not contain class files created during a previous run.

The Java Compiler API (JSR 199) provides a set of interfaces that describes the functions provided by a Java compiler and a service provider framework. Because vendors can provide their own implementations of these interfaces, you must run the JCK compiler tests against each available compiler by using all possible interfaces.

You must use the JCK compiler interview to specify how each the tested compiler is run. In the single-VM mode, tests for the available compiler must be run through the Java Compiler API. In the multi-VM mode, tests for the available compiler can be run either through command-line interface or through the Java Compiler API.

Note – If your platform has an alternative compiler or compilers available through the Java Compiler API and a service provider framework, use the interview to specify the compiler that is tested. To do this, set the Default Compiler answer to No and enter the alternative compiler class name in the Compiler Class Name interview question. When the test is run, the harness selects the compiler by its name from the list of available compilers.

5.4.2 Configuration

To enable new language features (such as generics) introduced in Java 2 Platform Standard Edition 5.0, you might be required to answer the Other Options question in each interview. For example, the JDK software compiler requires the `-source 1.5` option to enable new language features.

5.4.3 Execution

No special requirements.

5.4.4 Configuration Tab Name-Value Pairs

The Configuration tab name-value pairs in Table 5–7 are provided for use with the JavaTest harness test view and Test Environment dialog box when troubleshooting problems with running tests. You can view the configuration values that were used to run a specific test by choosing the Configuration tab in the Test Manager.

For a description of the test view tabs and Test Environment dialog box, see the *JavaTest Harness User's Guide: Graphical User Interface* included with this release or the JavaTest harness online help.

If you intend to set a configuration value in the command line to run tests, see Chapter 4 for a description of how to obtain the required question-tag names for your configuration.

Table 5–7 lists the name-values pairs displayed in the Configuration tab for the Compiler tests.

TABLE 5–7 Compiler Test Configuration Tab Name-Value Pairs

Name	Value Description
Select Compiler	The way to run a compiler. In the multi-VM mode it can be run either through the command-line interface or through the Java Compiler API. Example Configuration tab value: `Java Compiler API`
Java Launcher (only for multi-VM mode and Compiler API)	This is the command used to invoke the Java runtime system used to run your Java compiler. Example Configuration tab value: `jdk_install_dir/bin/java`
Runtime Classpath (only for multi-VM mode and Compiler API)	The way to pass the classpath for the Java runtime system used to run your Java compiler. Can be set either as an environment variable or as a command-line option. **Note –** This value determines the subsequent set of name-value pairs.
Classpath Environment Variable (only for multi-VM mode and Compiler API)	The name of the environment variable used to set the classpath. Example Configuration tab value: `CLASSPATH`
Classpath Option (only for multi-VM mode and Compiler API)	The option template for use in setting the classpath. Example Configuration tab value: `-classpath`
Other Environment Variables (only for multi-VM mode and Compiler API)	Other environment variables for Java runtime system used to run your Java Compiler. Example Configuration tab value: `name1=value1`
Other Options	Any other command-line options that must be set for your Java compiler. Example Configuration tab value: `-source 1.6`

TABLE 5–7 Compiler Test Configuration Tab Name-Value Pairs *(Continued)*

Name	Value Description
Default Compiler	A switch between the default compiler provided by `ToolProvider.getSystemJavaCompilerTool()` and another compiler made available through the service provider mechanism. Example Configuration tab value: `No`
Compiler Class Name	An alternative compiler class name. Example Configuration tab value: `com.vendor.Compiler`

5.5 CORBA Tests

Table 5–8 provides the location of the CORBA tests and identifies the API that is tested.

TABLE 5–8 CORBA Test Information

Test Information	Description or Value
Area tested	CORBA API
Test URL	`api/org_omg` `api/javax_rmi`

These tests test IDL to Java library code mapping, ORB features, and the CosNaming name server.

5.5.1 Setup

Before you execute the CORBA tests, you must start the Transient Name Service (TNS). To run network tests, you must also set up a remote host for communication with the CORBA tests during the test run.

▼ Starting the Transient Name Service

Before you execute the tests, you must start the TNS. The TNS is located in the `bin` directory of the JDK release.

The following commands start the server listening on the default port (900).

- **To start the TNS on the Solaris platform, type the following command:**

  jdk`/bin/tnameserv`

  Note – On the Solaris platform, you must have root access to use port 900.

- **To start the TNS on Win32, type the following command:**

  jdk`\bin\tnameserv`

  You can start the server listening on a different port.

- **To use a port other than the default port, include the** `-ORBInitialPort` **flag and a port number in the command used to start the TNS.**

  For example, type the following command to use port 1234:

  ```
  jdk/bin/tnameserv -ORBInitialPort 1234
  ```

- **Use the following command to change the port value in the** `.jti` **file so that it reflects the new port number:**

  ```
  OrbPortID=1234
  ```

Setting up the CORBA Distributed Test Run

Many CORBA tests communicate with a remote host during the test run. The remote host consists of a passive agent running on a Sun reference Java virtual machine (reference VM) located on a system other than the one being tested.

Note – The reference VM for the remote agent must be the same version as the VM under test.

Follow these steps to run network tests:

1. **Set up and start the distributed test host.**
2. **Copy** `javatest.jar` **to the distributed test server.**
3. **With** `javatest.jar` **on the classpath, start either the GUI or command-line version of the passive agent on the distributed test server.**

   For detailed information about starting a JavaTest harness agent, see the *JavaTest Agent User's Guide*.

Note – By default, the agent uses port 1908. If that port is not available, you can change it using the `-passivePort` option. Be sure that the port you use matches the one you specify in the `remote.networkAgent` configuration value.

4 **Run the CORBA tests.**

5.5.2 Configuration

A number of the CORBA tests use a distributed framework for testing the CORBA communications.

5.5.2.1 Distributed Tests

Distributed tests validate API functionality while the test platform is connected to a remote host. One part of the distributed test runs as usual on the platform under test and the other part of the test runs on a remote host using a passive agent running on a reference VM. For these tests to run, use each interview and its instructions to configure the JavaTest harness for using a passive agent. After the interview is complete, the passive agent must be started on the remote host as described in "Setting up the CORBA Distributed Test Run" on page 121.

5.5.3 Execution

No special requirements.

5.5.4 Configuration Tab Name-Value Pairs

The Configuration tab name-value pairs in Table 5–9 are provided for use with the JavaTest harness test view and Test Environment dialog box when troubleshooting problems with running tests. You can view the configuration values that were used to run a specific test by choosing the Configuration tab in the Test Manager.

For a description of the test view tabs and Test Environment dialog box, see the *JavaTest Harness User's Guide: Graphical User Interface* included with this release or the JavaTest harness online help.

If you intend to set a configuration value in the command line to run tests, see Chapter 4 for a description of how to obtain the required question-tag names for your configuration.

Table 5–9 lists the name-values pairs displayed in the Configuration tab for the CORBA tests.

TABLE 5–9 CORBA Test Configuration Tab Name-Value Pairs

Name	Value Description
`ORBHost`	The name of a host or machine running the TNS. You can use the alias `localhost` to specify the currently running host. Example Configuration tab value: `localhost`
`OrbPortID`	The port number listened to by the TNS.

5.6 Distributed Tests

Many network, Java APIs for XML Web Services (JAX-WS), Java Debug Wire Protocol (JDWP), and Java Management Extensions (JMX™ API) Remote tests communicate with a remote host during the test run. The remote host consists of a passive agent running on one of the Sun reference VMs on a system other than the one being tested.

Note – The reference VM for the remote agent must be the same version as the VM under test.

5.6.1 Setup

Before you can run the distributed tests, you must set up a remote host with a passive agent.

▼ Setting up the Distributed Tests Run

1. **Disable network data and packet filtering software.**
   This requirement includes Personal Firewall provided with Windows XP.
2. **Set up and start the distributed test host.**
3. **Verify that the test suite installation directory is reachable from a remote computer through the network drive or NFS mount.**
   The JAX-WS tests must use files located in the test suite installation directory.
4. **Start the passive agent on the distributed test server.**

Note – The passive agent for distributed tests must be run with the JCK security policy in effect. The reference VM running the passive agent must be started with the `java.security.policy` set. For example, the following VM option must be specified when running the passive agent:

`-Djava.security.policy=`*JCK*`/lib/jck.policy`

---

a. **Copy `javatest.jar` to the distributed test server.**

b. **With `javatest.jar` on the classpath, start either the GUI or command-line version of the passive agent from a command prompt.**

   See the *JavaTest Agent User's Guide* for detailed information about starting JavaTest harness agents.

   Note – By default the agent uses port 1908. If that port is not available, you can change it using the `-passivePort` option. Be sure that the port you use matches the one you specify in the `remote.networkAgent` configuration value.

Refer to "5.6.2 Configuration" on page 124 for information about configuring the JavaTest harness for using a passive agent to run Distributed Tests.

5.6.2 Configuration

Many network and JDWP tests use a distributed framework for testing the network API and the JDWP implementation. The distributed tests validate technology functionality while the test platform is connected to a remote host. One part of the distributed test runs on the platform under test and the other part of the test runs on a remote host using a passive agent. Always run the agent on a reference VM.

The JAX-WS tests use files from the test suite installation directory. To run JAX-WS tests, you must use a map file on a Remote Agent to map the test suite installation directory file syntax between systems. Refer to Appendix B, "Detailed Examples," the *JavaTest Agent User's Guide*, or the JavaTest harness online help for detailed instructions about creating and using map files.

You must complete the Remote Agent section of each interview to configure the JavaTest harness for using a passive agent. The Remote Agent section of each interview begins with the question titled "Remote Agent Configuration." After the interview is completed, start the passive agent on the remote host as described in the *JavaTest Agent User's Guide*.

If you have problems running the tests with the passive agent, use Table 5–10 to verify the configuration settings of the agent in the JavaTest harness.

5.6.3 Execution

See "5.11 JAX-WS Mapping Tests" on page 136, "5.13 JDWP Tests" on page 141, and "5.25 Network Tests" on page 193for detailed information about specific test execution.

5.6.4 Configuration Tab Name-Value Pairs

The Configuration tab name-value pairs in Table 5–10 are provided for use with the JavaTest harness test view and Test Environment dialog box when troubleshooting problems with running tests. You can view the configuration values that were used to run a specific test by choosing the Configuration tab in the Test Manager.

If you have problems running the tests with the passive agent, use Table 5–10 to verify the name-value settings of the agent in the JavaTest harness.

For a description of the test view tabs and Test Environment dialog box, see the *JavaTest Harness User's Guide: Graphical User Interface* included with this release or the JavaTest harness online help.

If you intend to set a configuration value in the command line to run tests, see Chapter 4 for a description of how to obtain the required question-tag names for your configuration.

Table 5–10 lists the name-values pairs displayed in the Configuration tab for the Distributed tests.

TABLE 5–10 Distributed Test Configuration Tab Name-Value Pairs

Name	Value Description

`remote.networkAgent`	Specifies the parameters required to connect with the agent running on the remote host:
	`-host` *host-name*
	Contacts the agent on host *host-name*.
	The following parameter is required:
	`-port` *port-number*
	Contacts the agent on port *port-number*. If omitted, the default port (1908) is used.
	The following parameter is optional:
	`-classpath` *path*
	Loads the classes to be executed on the remote host from *path*. If omitted, the agent loads classes from the agent's system classpath
	Example Configuration tab value:
	`-host localhost \`
	`-classpath $testClassDir:\`
	`$testSuiteRootDir/.../classes`

5.7 Extra-Attribute Tests

Table 5–11 provides the location of the Extra-Attribute (ATR) tests and identifies the areas that are tested.

TABLE 5–11 Extra-Attribute Test Information

Test Information	Description or Value
Area tested	Additional attributes in class files
Test URLs	`vm/classfmt/atr`

5.7.1 Setup

Because these tests use C code, it is not possible to define their compilation in a platform-independent way. Therefore, you must compile these tests before running them. These files must be compiled into a library named `jckatr` for loading using the method

`System.loadLibrary` (`"jckatr"`). To build the `jckatr` library, compile the file *jck*`/src/share/lib/atr/jckatr.c`, that contains a list of `#include` statements that refer to other files in the *jck*`/tests` directory.

When building the `jckatr` library, if the library is linked dynamically, you must set up a platform-dependent system environment variable such as `LD_LIBRARY_PATH` on the Solaris platform, or `PATH` on Win32 to load the `jckatr` libraries.

See "Building `jckatr.dll` for Win32 Systems" on page 127, "Building `libjckatr.so` With C Compiler for Solaris Platform" on page 128, and "Building `libjckatr.so` With Gnu C for Solaris Platform" on page 129for the procedures required to build the `jckatr` library on Win32 systems and the Solaris platform.

▼ Building `jckatr.dll` for Win32 Systems

- **Use the command lines in Example 5–1 to build the `jckatr.dll` library for Win32 systems using the Microsoft Visual C++ (MSVC++) compiler.**

  Substitute the exact JCK and destination paths for the strings *jck* and *dest*.

Example 5–1 Using MSVC++ Compiler to Build `jckatr.dll`

```
cl /LD /MD /TP /DWIN32 /DNDEBUG /D_WINDOWS /D_USRDLL \
/D_WINDLL /D_AFXDLL /Ijck /Fodest\jckatr.obj \
jck\src\share\lib\atr\jckatr.c \
/link /PDB:NONE /DLL /DEF:jck\src\share\lib\atr\jckatr.def \
/OUT:dest\jckatr.dll
```

5.7.1.1 Command-Line Options for MSVC++ Compiler

Table 5–12 contains the command-line options used to build the `jckatr.dll` library for Win32 systems.

TABLE 5–12 Command-Line Options for MSVC++ Compiler

Option	Description
`cl`	The MSVC++ compiler.
`/LD /MD`	Create `.DLL` and link it with `MSVCRT.LIB` (multi-threaded runtime).
`/TP`	Compile all files as C++ sources.
`/DWIN32`	Define WIN32 for C preprocessor.
`/D_WINDOWS`	Define `_WINDOWS` for C preprocessor.

TABLE 5–12 Command-Line Options for MSVC++ Compiler *(Continued)*

Option	Description
`/I`*jck*	Add the directory to include search path.
`/Fo`*dest*	Specify the intermediate object file.
jck`\src\share\lib\atr\jckatr.c`	Input file.
`/link`	Linker options follow.
`/PDB:NONE`	Do not create a Program Database.
`/DLL`	Build a `dll`.
`/DEF:`*jck*`\src\share\lib\atr\jckatr.def`	Module-definition file.
`/OUT:`*dest*`\jckatr.dll`	Specify the output file name. *dest* represents the path of the destination directory.

Note – To use the MSVC++ compiler, you might need to set up some environment variables, such as `INCLUDE`, `LIB`, and `PATH`. Use the file `vcvars32.bat` located in the installed `bin` directory (from MSVC++) to set up environment variables.

▼ Building `libjckatr.so` With C Compiler for Solaris Platform

- **Use the command lines from Example 5–2 to build the library for the Solaris platform.**
  Substitute the exact JCK and destination paths for the strings *jck* and *dest*.

Example 5–2 Using C Compiler to Build `libjckatr.so`

```
cc -G -KPIC -o dest/libjckatr.so -Ijck jck/src/share/lib/atr/jckatr.c
```

5.7.1.2 Command-Line Options for the Solaris Platform C Compiler

Table 5–13 contains the command-line options used to compile the test files for the Solaris platform.

TABLE 5–13 Command-Line Options for the Solaris Platform C Compiler

Option	Description
`cc`	C compiler.
`-G`	Generate a shared library.
`-KPIC`	Generate position-independent code.

TABLE 5–13 Command-Line Options for the Solaris Platform C Compiler *(Continued)*

Option	Description
`-I`*jck*	Add the JCK directory to the `include` search path.
`-o` *dest*`/libjckatr.so`	Specify the output file name. *dest* represents the path of the destination directory.
jck`/src/share/lib/atr/jckatr.c`	Input file.

▼ Building `libjckatr.so` With Gnu C for Solaris Platform

- **Use the command lines from Example 5–3 to build the library using the Gnu C compiler for the Solaris platform.**

  Substitute the exact JCK and destination paths for the strings *jck* and *dest*.

Example 5–3 Using Gnu C Compiler to Build the `libjckatr.so`

```
gcc -fPIC -shared -o dest/libjckatr.so -Ijck jck/src/share/lib/atr/jckatr.c
```

5.7.1.3 Command-Line Options for the Gnu C Compiler for the Solaris Platform

Table 5–14 contains the command-line options used to build the library using the Gnu C compiler for the Solaris platform.

TABLE 5–14 Command-Line Options for the Gnu C Compiler for the Solaris Platform

Option	Description
`gcc`	Gnu C Compiler.
`-fPIC`	Emit position-independent code, suitable for dynamic linking.
`-shared`	Produce a shared object.
`-o` *dest*`/libjckatr.so`	Specify the output file name. *dest* represents the path of the destination directory.
`-I`*jck*	Add the JCK directory to the `include` search path.
jck`/src/share/lib/atr/jckatr.c`	Input file.

5.7.2 Configuration

No special requirements.

5.7.3 Execution

No special requirements.

5.7.4 Configuration Tab Name-Value Pairs

The Configuration tab name-value pairs in Table 5–15 are provided for use with the JavaTest harness test view and Test Environment dialog box when troubleshooting problems with running tests. You can view the configuration values that were used to run a specific test by choosing the Configuration tab in the Test Manager.

For a description of the test view tabs and Test Environment dialog box, see the *JavaTest Harness User's Guide: Graphical User Interface* included with this release or the JavaTest harness online help.

If you intend to set a configuration value in the command line to run tests, see Chapter 4 for a description of how to obtain the required question-tag names for your configuration.

Table 5–15 lists the name-values pairs displayed in the Configuration tab for the extra-attribute tests.

TABLE 5–15 Extra-Attribute Test Configuration Tab Name-Value Pairs

Name	Value Description
`platform.nativeCodeSupported`	If your system supports native code, this value is `true`. This value signifies that the VM under test provides support for loading native code. If your system does not provide support for loading native code, this value must be `false`.
`PATH` `LD_LIBRARY_PATH`	The `.jti` file created by the Configuration Editor must include any necessary platform-specific variables used to identify the path to the library file. On Win32 systems, the variable is `PATH`. On the Solaris platform, the variable is `LD_LIBRARY_PATH`. If you execute your tests using the JavaTest harness agent, you must set the platform-specific variables (`PATH` or `LD_LIBRARY_PATH`) before you start the agent. See *jck*`/win32/bin/javatest` for a useful script.

5.8 Floating-Point Tests

Table 5–16 provides the location of the floating-point tests and identifies the areas that are tested.

TABLE 5–16 Floating-Point Test Information

Test Information	Description or Value
Areas tested	Virtual machine Language
Test URLs	`vm/fp` `lang/FP`

JCK includes several tests for floating-point calculations. Extended floating-point formats might be used to represent the values of `float` and `double` data types in the default mode of calculations. The platform-specific parameters of these extended floating-point formats are required for some of the tests to execute successfully.

5.8.1 Setup

No special requirements.

5.8.2 Configuration

No special requirements.

5.8.3 Execution

No special requirements.

5.8.4 Configuration Tab Name-Value Pairs

The Configuration tab name-value pairs in Table 5–17 are provided for use with the JavaTest harness test view and Test Environment dialog box when troubleshooting problems with running tests. You can view the configuration values that were used to run a specific test by choosing the Configuration tab in the Test Manager.

For a description of the test view tabs and Test Environment dialog box, see the *JavaTest Harness User's Guide: Graphical User Interface* included with this release or the JavaTest harness online help.

If you intend to set a configuration value in the command line to run tests, see Chapter 4 for a description of how to obtain the required question-tag names for your configuration.

Table 5–17 lists the name-values pairs displayed in the Configuration tab for the floating-point tests.

TABLE 5–17 Floating-Point Test Configuration Tab Name-Value Pairs

Name	Value Description

`hardware.xFP_ExponentRanges`	Four numbers that describe the ranges of exponents that can be represented in extended-exponent formats of floating-point values. The numbers are ordered and separated by a colon (:). The first two numbers denote the minimum and maximum exponent representable in extended-exponent format for the type `float`. The last two numbers denote the minimum and maximum exponent representable in extended-exponent format for the type `double`.
	
	The following example describes parameters for the Intel x86 80-bit `float` and `double` formats. Both formats provide 15 bits for exponents.
	
	Example Configuration tab value:
	
	`-16382:16383:-16382:16383`
	
	If the Java technology implementation under test does not use extended-exponent floating-point formats in non-`strictfp` calculations, the value must be equal to the parameters of standard floating-point formats shown in the following example.
	
	Example Configuration tab value:
	
	`-126:127:-1022:1023`

For JCK runtime, the default value in each interview corresponds to the standard formats of the Java programming language `float` and `double` types. If your implementation uses extended floating-point formats, you must set this value in the configuration.

For JCK compiler, the value set in each interview must be appropriate for the JRE implementation used to execute the resulting class files. Sun provides reference implementations for the SPARC processor architecture and for the Intel x86 processor architecture. The default configuration value in each interview is for the SPARC processor reference architecture. If you use a reference implementation for the Intel x86 processor architecture, you must also set the value for the Intel x86 reference architecture in the JCK configuration file.

5.9 ImageIO Tests

Table 5–18 provides the location of the Java ImageIO API tests and identifies the area that is tested.

TABLE 5–18 ImageIO Test Information

Test Information	Description or Value
API tested	`javax.imageio`
Test URLs	`api/javax_imageio/ImageIO`

5.9.1 Setup

Before using a browser to run Java ImageIO API tests, you must add the `ImageIO-resources.jar` file located at *URL-TO-JCK-ROOT*`/tests/api/javax_imageio/ImageIO` to the archive attribute in the HTML code used to launch the agent.

EXAMPLE 5–4 Applet Code to Launch a JavaTest Agent

```
<applet
code="com.sun.javatest.agent.AgentApplet"
codebase="[URL-TO-JCK-ROOT]/classes"
archive="../lib/javatest.jar,
../tests/api/javax_imageio/ImageIO/ImageIO-resources.jar"
width=600
height=600 >
</applet>
```

See Appendix B for detailed information about how to use the JavaTest harness with a browser (Netscape Navigator™ and JavaPlug-In software) and an agent to run JCK tests.

5.9.2 Configuration

No special requirements.

5.9.3 Execution

No special requirements.

5.9.4 Configuration Tab Name-Value Pairs

None.

5.10 Interactive Tests

Interactive tests require user interaction. All interactive tests in the JCK test suite are executed at one time, irrespective of the fact that they belong to different test API areas.

The following JCK test areas include interactive tests:

- `api/java_awt/interactive`
- `api/javax_print/interactive`
- `api/javax_sound/interactive`
- `api/javax_swing/interactive/`

Interactive tests have been developed using two kinds of user interfaces:

- Yes-No interface — Tests that use the Yes-No interface request the user to perform a visual verification and keyboard-mouse actions, and to click Yes or No to declare whether the test passed or failed. If the user clicks No in the case of a failed test, the user is prompted to enter text that describes the failure. That text is included in the test's results file (`.jtr`).
- Done interface — Tests that use the Done interface request the user to perform keyboard-mouse actions and click Done.

5.10.1 Setup

Because interactive tests require keyboard and mouse input, you might decide to run them separately and not batched with the rest of the test suite.

Interactive tests contain the `interactive` or `robot` keyword. You can use the keywords field in the JavaTest Configuration Editor to filter or include interactive tests in a test run.

- `interactive` — All interactive tests contain this keyword. Use the `interactive` keyword to include interactive tests in a test run. Use the `!interactive` expression to exclude interactive tests from a test run.
- `robot` — Some AWT interactive tests contain the `robot` keyword. Use `robot` keyword if your platform supports the option to run interactive tests without human interaction, which is done by using the `java.awt.Robot` class. Tests that use the robot can be selected or deselected by specifying the `robot` keyword in the configuration. See "5.3 AWT and Swing Tests" on page 114 for more information about using the `robot` keyword.

For more information about using keywords in the configuration, see Chapter 4.

Running the interactive tests separately reduces the time required to find the tests that use the `interactive` or `robot` keywords. To run the interactive tests separately, specify in the configuration which interactive tests to run (for example, AWT and Swing, Printing, and Sound tests or just some of these). See Chapter 4 for detailed information.

Interactive tests time out if the user does not click Yes-No or Done within a certain amount of time. Timeout values for these tests vary. The timeout value for a test is based on the complexity of the test. A test must be rerun if it times out, perhaps with a different timeout value.

Note – You can use the Pause-Resume button in each interactive test frame to pause the test timer and then work with the test as long as you want.

For test-specific information, see respectively:

- "5.3 AWT and Swing Tests" on page 114
- "5.28 Printing Tests" on page 200
- "5.32 Sound Tests" on page 211

5.10.2 Configuration

For test-specific information, see respectively:

- "5.3 AWT and Swing Tests" on page 114
- "5.28 Printing Tests" on page 200
- "5.32 Sound Tests" on page 211

5.10.3 Execution

For test-specific information, see respectively:

- "5.3 AWT and Swing Tests" on page 114
- "5.28 Printing Tests" on page 200
- "5.32 Sound Tests" on page 211

5.10.4 Configuration Name-Pair Values

For test-specific information, see respectively:

- "5.3 AWT and Swing Tests" on page 114
- "5.28 Printing Tests" on page 200
- "5.32 Sound Tests" on page 211

5.11 JAX-WS Mapping Tests

Two kinds of JAX-WS tests are included in the devtools compiler test suite.

- Java-to-WSDL mapping tests do mapping `.java` classes onto WSDL files
- WSDL-to-Java mapping tests do mapping WSDL files with binding files onto `.java` classes

TABLE 5–21 provides the location of the JAX-WS mapping tests and identifies the areas that are tested.

TABLE 5–19 JAX-WS Mapping Test Information

Test Information	Description or Value
API tested	Not applicable. Tests for JAX-WS API are located in the JCK runtime test suite.
Test URLs	`jaxws`

5.11.1 Setup

Executing JAX-WS mapping tests is implementation-specific. Depending on whether you choose to execute in multi-VM or in single-VM mode, you must provide either a wrapper script or a wrapper class for JCK to do JAX-WS mapping.

If you choose to start a new instance of the product from a command line each time a new test is executed, the JCK invokes the JAX-WS mapping as a separate process for each test. The mapping and tests will run in multi-VM mode.

For detailed instructions required to provide and use a wrapper script to execute a JAX-WS mapping in this test environment, see "5.11.1.1 Running Java-to-WSDL Mapping in Multi-VM Mode" on page 136 and "5.11.1.2 Running WSDL-to-Java Mapping in Multi-VM Mode" on page 137.

If you choose not to start a new instance of the product from a command line each time a new test is executed, all mapping will be done and tests are executed in a single process, the agent's VM. For detailed instructions required to provide and use a wrapper class to execute a JAX-WS mapping in this test environment, see "5.11.1.3 Running Java-to-WSDL Mapping in Single-VM Mode" on page 138 and "5.11.1.4 Running WSDL-to-Java Mapping in Single-VM Mode" on page 138.

5.11.1.1 Running Java-to-WSDL Mapping in Multi-VM Mode

In multi-VM mode, Java-to-WSDL mapping is invoked as a separate process for each test. The JCK invokes the wrapper script for mapping using the `java.lang.Runtime.exec(String[] cmdarray, String[] envp)` method, where `cmdarray` is the following form:

[environment-variables] command `-cp` *classpath* `-d` *out-dir java-files*

Optional parameters are enclosed in square brackets ([]). The devtools compiler test suite assumes that with the `-d` option, an output directory is specified and the `.java` classes to map are specified at the end of command string.

The following variables are included in the command string:

- *environment-variables* - Environment variables if required can be specified in the Other environment variables question of Configuration Editor.
- *command* - The name of the wrapper script that does Java-to-WSDL mapping.
- *classpath* - The classpath constructed by the devtools test suite.
- *out-dir* - The directory in which the generated files are placed.
- *java-files* - The `.java` class files to map.

The devtools test suite provides two sample scripts that work for the reference JAX-WS Java-to-WSDL mapping. You can use the following scripts to develop scripts for other JAX-WS implementations:

- `solaris/bin/wsgen.sh` (or `linux/bin/wsgen.sh`) - A `ksh` script used to run tests with the reference JAX-WS mapping tool on Solaris or Linux platforms.
- `win32/bin/wsgen.bat` - A Windows batch script used to run tests with the reference JAX-WS mapping tool on Windows platforms. This script is similar to the `solaris/bin/wsgen.sh` script.

5.11.1.2 Running WSDL-to-Java Mapping in Multi-VM Mode

In multi-VM mode, WSDL-to-Java mapping is invoked as a separate process for each test. The JCK invokes the wrapper script for mapping by using the `java.lang.Runtime.exec(String[] cmdarray, String[] envp)` method, where `cmdarray` is the following:

[environment-variables] command `-cp` *classpath* `[-b` *binding-file*`] -d` *out-dir wsdl-files*

Optional parameters are enclosed in square brackets ([]). The devtools compiler test suite assumes that option `-b` denotes binding file, with `-d` option output directory is specified and `.java` classes to map are specified at the end of command string.

The following variables are included in the command string:

- *environment-variables* - Environment variables if required can be specified in the Other environment variables question of Configuration Editor.
- *command* - The wrapper script that do WSDL-to-Java mapping.
- *classpath* - The classpath is constructed by JCK-devtools test suite.
- *binding-file* - The binding file. Use of more than one binding file is allowed. Each binding file must be preceded by a `-b` option.

- *out-dir* - The directory in which the generated files are placed.
- *wsdl-files* - The WSDL files to map.

The devtools test suite provides two sample scripts that work for the reference WSDL-to-Java mapping. You can use the following scripts to develop scripts for other JAX-WS implementations:

- `solaris/bin/wsimport.sh` (or `linux/bin/wsimport.sh`) - A `ksh` script. Used to run tests with the reference JAX-WS mapping tool on Solaris or Linux platforms.
- `win32/bin/wsimport.bat` - Windows batch script. Used to run tests with the reference JAX-WS mapping tool on Windows platforms. This script is similar to the `solaris/bin/wsimport.sh` script.

5.11.1.3 Running Java-to-WSDL Mapping in Single-VM Mode

In single-VM mode, all invocations of a Java-to-WSDL mapping are made from the JavaTest agent running in the same VM. A wrapper class for the schema compiler must implement the following `com.sun.jck.lib.WSGenTool` interface.

```
package com.sun.jck.lib;
public interface WSGenTool {
int generate(String sei, File outDir, PrintStream out, PrintStream err);
}
public int compile_method(String[] args, java.io.PrintStream out,
java.io.PrintStream err)
```

Java-to-WSDL mapping should be accomplished during the invocation of the generate method.

- `sei` - `.java` file to be mapped
- `outDir` - Output directory where `wsdl` is generated
- `out` - Output stream for logging
- `err` - Error stream for logging

The JCK includes the following sample class that is fully compatible with Sun's reference implementation and that can be used for Java-to-WSDL mapping:

`com.sun.jck.lib.WSGen`

5.11.1.4 Running WSDL-to-Java Mapping in Single-VM Mode

In single-VM mode, all invocations of WSDL-to-Java mapping are made from the JavaTest agent running in the same VM. A wrapper class for the schema compiler must implement the following `com.sun.jck.lib.WSImportTool` interface:

```
package com.sun.jck.lib;
public interface WSImportTool {
int compile(String{} wsdlFiles, String[] xmlFiles, File outDir,
```

```
PrintStream out, PrintStream err);
}
```

WSDL-to-Java mapping should be accomplished during invocation of compile method.

- `wsdlFiles` - Array of strings containing `wsdl` files
- `xmlFiles` - Array of strings containing binding files
- `outDir` - Output directory where `.java` files will be generated
- `out` - Output stream for logging
- `err` - Error stream for logging

The JCK includes the following sample class that is fully compatible with Sun's reference implementation and that can be used as a WSDL-to-Java class:

`com.sun.jck.lib.WSImport`.

5.11.2 Configuration

JAX-WS tests use files from test suite installation directory. To run JAX-WS tests you must use a map file on a Remote Agent to map the test suite installation directory file syntax between systems. Refer to the *JavaTest Agent User's Guide*, the *JavaTest Harness User's Guide: Graphical User Interface*, or the JavaTest harness online help for detailed instructions about creating and using map files.

5.11.3 Execution

No special requirements.

5.11.4 Configuration Tab Name-Value Pairs

None.

5.12 JDBC Technology RowSet Tests

The JCK includes tests for implementation classes of five standard Java Database Connectivity (JDBC™) technology RowSet interfaces: `CachedRowSet`, `JdbcRowSet`, `WebRowSet`, `FilteredRowSet`, and `JoinRowSet`.

Table 5–20 provides the location of the JDBC technology RowSet tests and identifies the areas that are tested.

TABLE 5–20 JDBC TechnologyRowSet Test Information

Test Information	Description or Value
Area tested	Rowset implementation classes
Test URLs	`api/javax_sql/rowset/impl`

5.12.1 Setup

If you want to test your own rowset implementations, you must implement and compile the specific factory class that is used to create the rowsets. The class must have the public default constructor and implement the `javasoft.sqe.tests.api.javax.sql.rowset.impl.RowSetFactory` interface located at *jck*`/src/share/classes/javasoft/sqe/tests/rowset/util/RowSetFactory.java`.

This interface contains methods for creating the rowset instances. The method used must return a newly created instance or instances each time it is invoked.

For example, to compile the custom factory with Sun's Java compiler, use the following command:

jdk`/bin/javac -cp` *jck*`/classes -d out CustomRowSetFactory.java`

Where:

- *jdk* - Path to Sun's JDK software
- *jck*`/classes` - Path to the TCK's JDBC technology RowSet classes
- `out` - Classes output directory
- `CustomRowSetFactory.java` - Factory provided by the user

The location of the compiled factory class or classes must be added to the JCK classpath. For multi-VM mode, you must add the location in the answer to the Additional Classpath interview question. For a single-VM mode, you must add the corresponding classpath to the JavaTest agent invocation command.

For testing reference implementation of rowsets the `javasoft.sqe.tests.api.javax.sql.rowset.impl.RowSetFactoryImpl` is used. This class is located in the *jck*`/classes` directory.

5.12.2 Execution

No special requirements.

5.12.3 Special Configuration Steps

No special requirements.

5.12.4 Configuration Tab Name-Value Pairs

The Configuration tab name-value pairs in Table 5–21 are provided for use with the JavaTest harness test view and Test Environment dialog box when troubleshooting problems with running tests. You can view the configuration values that were used to run a specific test by choosing the Configuration tab in the Test Manager.

For a description of the test view tabs and Test Environment dialog box, see the *JavaTest Harness User's Guide: Graphical User Interface* included with this release or the JavaTest harness online help.

If you intend to set a configuration value in the command line to run tests, see Chapter 4 for a description of how to obtain the required question-tag names for your configuration.

Table 5–21 lists the name-values pairs displayed in the Configuration tab for the JDBC technology Rowset tests.

TABLE 5–21 JDBC Technology Rowset Tests Configuration Tab Name-Value Pairs

Name	Value Description
`platform.rowSetFactory`	The fully qualified name of the factory class used for creating the rowsets.

5.13 JDWP Tests

The JCK includes tests for Java Debug Wire Protocol (JDWP). JDWP is the protocol used for communication between a debugger and the VM that it debugs, such as a target VM running a program.

Table 5–22 provides the location of the JDWP tests and identifies the areas that are tested.

TABLE 5–22 JDWP Test Information

Test Information	Description or Value
Area tested	Java Debug Wire Protocol
Test URLs	`vm/jdwp`

5.13.1 Setup

Before you can run JDWP tests, you must set up the test system. The following procedures are required to establish a transport class and location for the remote agent to load classes from the JavaTest harness.

▼ Setting up a Test System to Run JDWP Tests

1 Provide `TransportService` class.

A JDWP implementation might be represented either as a Java Virtual Machine Tool Interface (JVM TI) agent or as an integral part of a VM. JDWP tests use a pluggable transport mechanism for establishing a connection between debuggers and the JDWP implementation. The JCK provides a Java technology TCP socket-based transport. If you need a non-TCP-based transport for testing a JDWP implementation configuration, you must provide your own transport class.

The transport class provided must be a subclass of the `com.sun.jdi.connect.spi.TransportService` abstract class.

All required source files of `.java` classes from package `com.sun.jdi.connect.spi` are located in the following directory:

jck`/src/share/classes/javasoft/sqe/jck/lib/jpda/jdi/`

If you provide your own transport class, you must use either Sun's Java compiler or another certified compiler to compile it on one of the reference platforms.

Compiled class files of `com.sun.jdi.connect.spi` are located in corresponding subdirectories of the *jck*`/classes` directory.

Example compilation of provided transport class (using JDK software on the Solaris platform):

jdk`/bin/javac -d . -classpath` *jck*`/classes` *YourTransportServiceClass*`.java`

2 Set up a transport class location.

If you have configured a remote agent to load classes from the JavaTest harness, copy the compiled transport class into a directory accessible from the system where you start the JavaTest harness. Each interview asks you to specify a directory or a `.jar` file where the class can be found by the JavaTest harness. Otherwise, you must ensure that the transport class is in the remote agent's classpath.

3 Set up JDWP distributed test run.

All JDWP tests communicate with a remote host during the test run. The remote host consists of a passive agent running on one of the Sun reference VMs on a system other than the one being tested.

Note – The reference VM for the remote agent must be the same version as the VM under test.

a. **Set up and start the distributed test host.**

b. **Start the passive agent on the distributed test server:**

   i. **Copy `javatest.jar` to the distributed test server.**

   ii. **With `javatest.jar` on the classpath, start either the GUI or command-line version of the passive agent.**

   iii. **If the remote agent is configured to not load classes from the JavaTest harness and you provide your own transport class, you must provide that compiled transport class in the agent's classpath.**

See the *JavaTest Agent User's Guide* for detailed information about starting a JavaTest harness agent.

Note – If you only run the JDWP tests, you might start the remote agent on the system where the tested Java technology-based application is executed. If you run both JDWP and network API tests, you must start the remote agent on a system other than the one being tested. If the provided transport class only works on a specific platform, you might start the remote agent on that platform only to execute the JDWP tests.

5.13.2 Configuration

JDWP tests are distributed tests that validate JDWP functionality while the test platform is connected to a remote host. Normally, one part of a distributed test runs on the platform under test and the other part of the test runs on a remote host using a passive agent running on a reference VM. To run these tests, use each interview to configure the JavaTest harness for using a passive agent. After you complete the configuration, start the passive agent on the remote host as described in "5.13.1 Setup" on page 142.

Note – In some cases, the remote agent might be started on a machine if it is not running the reference VM. See "5.13.1 Setup" on page 142 for the description of this exception.

5.13.3 Execution

The JDWP tests are distributed tests. Debuggers are executed on the remote JavaTest agent. Debuggees are executed with the activated JDWP implementation on the tested platform.

The JDWP and the TransportService class specifications do not require restoring the JDWP transport connection after the debugging session ends. JDWP implementations might or might not start listening or attaching to debuggers after the previous debugging session is completed.

If the JCK is configured to execute tests in single-VM mode and the tested JDWP implementation does not restore connections after a debugging session ends, you must run each JDWP test separately. After each JDWP test completion, you must take the following actions on the tested platform with the activated JDWP implementation:

1. Kill the JavaTest agent process running the JavaTest harness agent.
2. Restart the JavaTest agent for execution of the next test.

The debugging parts of JDWP tests operate with timeout values defined by JavaTest harness time factor set in the interview. You must increase the time factor value in the interview if you experience test timeout problems on slow tested platforms.

Activating the JDWP implementation in a tested runtime might significantly affect non-JDWP JCK tests execution, therefore only JDWP tests must be executed on a tested runtime with activated JDWP implementation.

In multi-VM mode, additional settings are not required. The JavaTest harness passes JDWP-specific options to tested VMs when the jdwp keyword is in the test description.

However, in single-VM mode, only JDWP tests must be executed for testing JDWP implementation. Use keyword-based selection criteria (`!jdwp`) to run all but JDWP JCK tests. Use keyword-based selection criteria (`jdwp`) in addition to standard keyword selection criteria to run only JDWP JCK tests in single-VM mode.

5.13.4 Configuration Tab Name-Value Pairs

The Configuration tab name-value pairs in Table 5–23 are provided for use with the JavaTest harness test view and Test Environment dialog box when troubleshooting problems with running tests. You can view the configuration values that were used to run a specific test by choosing the Configuration tab in the Test Manager.

For a description of the test view tabs and Test Environment dialog box, see the *JavaTest Harness User's Guide: Graphical User Interface* included with this release or the JavaTest harness online help.

If you intend to set a configuration value in the command line to run tests, see Chapter 4 for a description of how to obtain the required question-tag names for your configuration.

Table 5–23 lists the name-values pairs displayed in the Configuration tab for the JDWP tests.

TABLE 5–23 JDWP Test Configuration Tab Name-Value Pairs

Name	Value Description
`platform.jdwpSupported`	Boolean value that indicates whether a JDWP implementation is provided for testing. If a JDWP implementation is provided, this value must be `true`.
`jdwpConnectorType`	Tested JDWP implementation configuration provides that the JDWP implementation either attaches or listens for connections from `debuggers`. The `jdwpConnectorType` value defines the debugger attaching-listening mode. If the tested JDWP implementation configuration provides that JDWP implementation listens for connections from debuggers, the value of `jdwpConnectorType` is attaching. Otherwise, it is listening.
`jdwpTransportClass`	The `jdwpTransportClass` defines the `TransportService` implementation class used by debuggers for establishing connection with tested JDWP implementation.
`jdwpTransportClassDir`	If you provide your own `TransportService` class implementation and the remote agent is configured to load classes from the JavaTest harness, `jdwpTransportClassDir` defines a directory or JAR file used by the JavaTest harness for loading your transport class. The `jdwpTransportClassDir` value has the following syntax: *pathSeparator directory* In this syntax, *pathSeparator* is a path separator string specific for the system where the JavaTest harness runs. For example, if you run the JavaTest harness on a Windows system, it is a semicolon (;) and on a Solaris platform it is a colon (:). In this syntax, *directory* is either the directory or the JAR file used by the JavaTest harness to search for your transport class.
`jdwpTransportAddress`	Transport-specific string that defines the address that test debuggers should use for establishing connections with JDWP implementation. For example, in the JCK TCP socket-based transport class, the address has the format: *host:port number* In this syntax, *host* and *port number* define host and port number to connect to the JDWP implementation. If JDWP tests are configured for debuggers to listen for connections from JDWP implementation, `jdwpTransportAddress` is *port number*, the `int` value representing the port number for the tests' debuggers to listen for connections from the JDWP implementation.

TABLE 5–23 JDWP Test Configuration Tab Name-Value Pairs *(Continued)*

Name	Value Description
`VMSuspended`	Boolean value that indicates whether or not the target VM with the activated JDWP implementation is suspended when initialized. If the VM is suspended, this value must be `true`.
`platform.MultiVM`	Boolean value that indicates whether or not the JCK is configured to execute tests in multi-VM mode.
`jdwpOpts`	Defines the Java platform command-line options specified in the tested command line used to activate JDWP and define the tested JDWP implementation configuration. This value is defined only if the JCK is configured to execute tests in multi-VM mode (such as `platform.multiJVM` is `true`). If you execute tests in single-VM mode, specify these options in the JavaTest harness agent start command line.

5.14 JMX API Advanced Dynamic Loading Tests

Table 5–24 provides the location of Advanced Dynamic Loading tests for Java Management Extensions (JMX) API and identifies the area that is tested.

TABLE 5–24 JMX API Advanced Dynamic Loading Test Information

Test Information	Description or Value
Area tested	JMX API, Advanced Dynamic Loading
Test URLs	`api/javax_management/loading`

5.14.1 Setup

The JMX API tests for advanced dynamic loading use C code to check loading of native libraries and cannot be compiled in a platform-independent way. Before running the JMX API tests, you must manually make the libraries for the platforms that you are using. If your implementation supports loading native libraries from JAR files, you must prepare two JAR files that contain the native libraries compiled for your platform.

▼ Setting up JMX API Tests on a Platform

1 **Copy HTML and JAR files from *jck*`/tests/api/javax_management/loading/data` to a location accessible during the test run.**

This step is optional if you are running tests on one platform. See "Copying Required HTML and JAR Files" on page 147 for detailed instructions.

2. **Make `systemInfo`, `jmxlibid`, and `genrandom` libraries.**

   See "Making `systemInfo`, `jmxlibid`, and `genrandom` Libraries" on page 147 for detailed instructions.

3. **Put `systemInfo` library into *dest*`/archives/MBeanUseNativeLib.jar`.**

   See "Updating *res-dest*`/archives/MBeanUseNativeLib.jar` on the Solaris Platform" on page 152 and "Updating *res-dest*`/archives/MBeanUseNativeLib.jar` on Win32 Systems" on page 152 for detailed instructions.

4. **Create a new *dest*`/archives/OnlyLibs.jar` file containing `jmxlibid` library. See "Creating *res-dest*`/archives/OnlyLibs.jar` on the Solaris Platform" on page 153 and "Creating *res-dest*`/archives/OnlyLibs.jar` on Win32 Systems" on page 153 for detailed instructions.**

Note – In the following examples, *jck* represents the JCK path (for example, `/java/re/jck60/JCK-runtime-60`), *dest* represents the destination path (for example, `/java/re/jck60/JCK-runtime-60/lib` or `/tmp/jmxresources`), *java* represents the path of the reference JDK software for the particular platform (for example, `/java/re/jdk60/solaris-sparc`), and *res-dest* represents the directory where you copy HTML and JAR files at the first step (for example, `/tmp/work/solaris-MultiVM`).

▼ Copying Required HTML and JAR Files

- **Copy HTML and JAR files from *jck*`/tests/api/javax_management/loading/data` to a location accessible during the test run.**

  This is required if you use the test suite to run the tests on various platforms. Each platform has a different native library. For example, to copy resources for JMX API tests on a Solaris platform, use:

  `cp -r` *jck*`/tests/api/javax_management/loading/data/*` *res-dest*

  This directory contains the library sources in the `archives/src` subdirectory. This directory is not needed for the test run. Do not copy it or remove it from the destination directory.

▼ Making `systemInfo`, `jmxlibid`, and `genrandom` Libraries

1. **Compile the `systemInfo`, `jmxlibid`, and `genrandom` libraries.**

   See "Compiling `libsystemInfo.so`, `libjmxlibid.so`, and `libgenrandom.so` for the Solaris Platform" on page 148 for instructions on compiling for the Solaris platform.

   See "Compiling `libsystemInfo.so`, `libjmxlibid.so`, and `libgenrandom.so` for the Solaris Platform Using Gnu C" on page 149 for instructions on using the Gnu C compiler when compiling for the Solaris platform.

See "Compiling `systemInfo.dll`, `jmxlibid.dll`, and `genrandom.dll` for Win32 Systems Using Microsoft Visual C++" on page 151 for instructions on compiling for a Win32 system.

For JMX API tests, the `systemInfo` and `jmxlibid` libraries must only exist in the JAR files.

2. **Compile the `systemInfo` and `jmxlibid` libraries to *jck*`/lib/libsystemInfo.so` and *jck*`/lib/libjmxlibid.so` (or `*.dll` for Win32 systems).**

3. **Place the compiled `systemInfo` and `jmxlibid` libraries into JAR files.**

Note – Do not leave `systemInfo` and `jmxlibid` libraries in a directory that can be accessed during a test run. For example, on the Solaris platform the libraries must *not* be located in the directory listed in `LD_LIBRARY_PATH`. Only the `genrandom` library must be available for loading. For example, on the Solaris platform, it must be located in the directory listed in `LD_LIBRARY_PATH`.

---

4. **Remove the `systemInfo` and `jmxlibid` libraries.**

▼ Compiling `libsystemInfo.so`, `libjmxlibid.so`, and `libgenrandom.so` for the Solaris Platform

- **Use the command lines in Example 5–5 to compile native libraries that are a part of JMX API tests for the Solaris platform.**

  Example 5–5 contains an example of compiling the `systemInfo`, `jmxlibid`, and `genrandom` libraries on the Solaris platform. Substitute the exact JCK and destination paths for the strings *jck* and *jest*.

Example 5–5 Compile JMX API Test Native Libraries for the Solaris Platform

```
cc -G -KPIC -o dest/libsystemInfo.so \
-Ijck/src/share/lib/jni/include \
-Ijck/src/share/lib/jni/include/solaris \
jck/tests/api/javax_management/loading/data/archives/src/C \
/com_sun_management_mbeans_loading_SystemInfoUseNativeLib.c

cc -G -KPIC -o dest/libjmxlibid.so \
-Ijck/src/share/lib/jni/include \
-Ijck/src/share/lib/jni/include/solaris \
jck/tests/api/javax_management/loading/data/archives/src/C \
/com_sun_management_mbeans_loading_GetLibIdFromNativeLib.c

cc -G -KPIC -o dest/libgenrandom.so \
-Ijck/src/share/lib/jni/include \
-Ijck/src/share/lib/jni/include/solaris \
```

```
jck/tests/api/javax_management/loading/data/archives/src/C \
/com_sun_management_mbeans_loading_RandomGen.c
```

5.14.1.1 Command-Line Options for Compiling Libraries on the Solaris Platform

Table 5–25 describes the available command-line options for compiling libraries on the Solaris platform.

TABLE 5–25 Command-Line Options for Compiling Libraries on the Solaris Platform

Option	Description
`cc`	C Compiler.
`-G`	Generate a shared library.
`-KPIC`	Generate position-independent code.
`-o` *dest*`/libsystemInfo.so`	Specify the output file name. The destination directory *dest* is usually *jck*`/lib`.
`-I`*jck*`/src/share/lib/jni/include`	Add the JCK `include` directory to the `include` search path.
`-I`*jck*`/src/share/lib/jni/include/solaris`	Add the JCK `include/solaris` directory to the `include` search path.

jck`/tests/api/javax_management/ \`
`loading/data/archives/src/C/ \`
`com_sun_management_mbeans_loading_`
`SystemInfoUseNativeLib.c`
	Input file.

▼ Compiling `libsystemInfo.so`, `libjmxlibid.so`, and `libgenrandom.so` for the Solaris Platform Using Gnu C

- **Use the command lines in Example 5–6 and the Gnu C compiler to compile native libraries that are a part of the JMX API tests for the Solaris platform.**

  Example 5–6 contains an example of using the Gnu C compiler to compile the `systemInfo`, `jmxlibid`, and `genrandom` libraries on the Solaris platform. Substitute the exact JCK and destination paths for the strings *jck* and *dest*.

Example 5–6 Compile Native Libraries for Solaris Platform Using the Gnu C Compiler

```
gcc -fPIC -shared -o dest/libsystemInfo.so \
-Ijck/src/share/lib/jni/include \
-Ijck/src/share/lib/jni//include/solaris \
jck/tests/api/javax_management/loading/data/archives/src/C \
/com_sun_management_mbeans_loading_SystemInfoUseNativeLib.c
```

```
gcc -fPIC -shared -o dest/libjmxlibid.so \
-Ijck/src/share/lib/jni/include \
-Ijck/src/share/lib/jni/include/solaris \
jck/tests/api/javax_management/loading/data/archives/src/C \
/com_sun_management_mbeans_loading_GetLibIdFromNativeLib.c

gcc -fPIC -shared -o dest/libgenrandom.so \
-Ijck/src/share/lib/jni//include \
-Ijck/src/share/lib/jni//solaris \
jck/tests/api/javax_management/loading/data/archives/src/C \
/com_sun_management_mbeans_loading_RandomGen.c
```

5.14.1.2 Command-Line Options for Compiling Libraries With GNU C Compiler on the Solaris Platform

Table 5–26 describes the command-line options for using the Gnu C compiler to compile libraries on the Solaris platform.

TABLE 5–26 Command-Line Options for Compiling Libraries With GNU C Compiler on the Solaris Platform

Option	Description
`gcc`	Gnu C Compiler.
`-fPIC`	Emit position-independent code, suitable for dynamic linking.
`-shared`	Generate a shared library.
`-o` *dest*`/libsystemInfo.so`	Specify the output file name. The destination directory *dest* is usually *jck*`/lib`.
`-I`*jck*`/src/share/lib/jni/include`	Add the JCK `include` directory to the `include` search path.
`-I`*jck*`/src/share/lib/jni/include/solaris`	Add the JCK `include/solaris` directory to the `include` search path.

jck`/tests/api/javax_management/\`
`loading/data/archives/src/C/\`
`com_sun_management_mbeans_loading_`
`SystemInfoUseNativeLib.c`
	Input file.

▼ Compiling `systemInfo.dll`, `jmxlibid.dll`, and `genrandom.dll` for Win32 Systems Using Microsoft Visual C++

- **Use the command lines in Example 5–7 and the Microsoft Visual C++ (MSVC++) compiler to build the libraries for Win32 systems.**

  Example 5–7 contains an example of compiling the `systemInfo.dll`, `jmxlibid.dll`, and `genrandom.dll` libraries on a Win32 system. Substitute the exact JCK and destination paths for the strings *jck* and *dest*.

Example 5–7 Build Win32 Libraries Using MSVC++ Compiler

```
cl /DWIN32 /D_WINDOWS /Ijck\src\share\lib\jni\include \
/Ijck\src\share\lib\jni\include\win32 \
/LD /MD /Fodest\systemInfo.obj /Fedest\systemInfo.dll \
jck\tests\api\javax_management\loading\data\archives\src\C\
com_sun_management_mbeans_loading_SystemInfoUseNativeLib.c

cl /DWIN32 /D_WINDOWS /Ijck\src\share\lib\jni\include \
/Ijck\src\share\lib\jni\include\win32 \
/LD /MD /Fodest\jmxlibid.obj /Fedest\jmxlibid.dll \
jck\tests\api\javax_management\loading\data\archives\src\C\
com_sun_management_mbeans_loading_GetLibIdFromNativeLib.c

cl /DWIN32 /D_WINDOWS /Ijck\src\share\lib\jni\include \
/Ijck\src\share\lib\jni\include\win32 \
/LD /MD /Fodest\genrandom.obj /Fedest\genrandom.dll \
jck\tests\api\javax_management\loading\data\archives\src\C\
com_sun_management_mbeans_loading_RandomGen.c
```

5.14.1.3 Command-Line Options for MSVC++ Compiler

Table 5–27 contains the command-line options used to build the libraries for Win32 systems.

TABLE 5–27 Command-Line Options for MSVC++ Compiler

Option	Description
`cl`	MSVC++ compiler.
`/LD /MD`	Create `.DLL` and link it with `MSVCRT.LIB` (multi-threaded runtime).
`/DWIN32`	Define WIN32 for C preprocessor.
`/D_WINDOWS`	Define `_WINDOWS` for C preprocessor.

TABLE 5–27 Command-Line Options for MSVC++ Compiler *(Continued)*

Option	Description
`/I`*jck*`\src\share\lib\jni\include` `/I`*jck*`\src\share\lib\jni\include\win32`	Add the directories to the `include` search path.
`/Fo`*dest*`\systemInfo.obj`	Specify the intermediate object file.
`/Fe`*dest*`\systemInfo.dll`	Specify the output file name.

jck`\tests\api\javax_management\loading\`
`data\archives\src\C\`
`com_sun_management_mbeans_loading_`
`SystemInfoUseNativeLib.c`
	Input file.

Note – To use the MSVC++ compiler, you might need to set up environment variables, such as `INCLUDE`, `LIB`, and `PATH`. You can use the `vcvars32.bat` batch file in the installed `bin` directory of MSVC++ to set up environment variables.

▼ Updating *res-dest*`/archives/MBeanUseNativeLib.jar` on the Solaris Platform

- **Use the following command to update the `MBeanUseNativeLib.jar` file on the Solaris platform.**

  jdk`/bin/jar uf` *res-dest*`/archives/MBeanUseNativeLib.jar systemInfo.so`

Note – Run this command from the directory containing the `libsystemInfo.so` file. Before running the command, make your current directory the directory containing the `libsystemInfo.so` file.

▼ Updating *res-dest*`/archives/MBeanUseNativeLib.jar` on Win32 Systems

- **Use the following command to update the `MBeanUseNativeLib.jar` file on Win32 Systems.**

  jdk`\bin\jar uf` *res-dest*`\archives\MBeanUseNativeLib.jar systemInfo.dll`

Note – Run this command from the directory containing the `systemInfo.dll` file. Before running the command, make your current directory the directory containing the `systemInfo.dll` file.

▼ Creating *res-dest*/archives/OnlyLibs.jar on the Solaris Platform

- **Use the following command to create the `OnlyLibs.jar` file containing `jmxlibid` library on the Solaris platform.**

  jdk/bin/jar cf *res-dest*/archives/OnlyLibs.jar libjmxlibid.so

---

Note – Run this command from the directory containing the `libjmxlibid.so` file. Before running the command, make your current directory the directory containing the `libjmxlibid.so` file.

▼ Creating *res-dest*/archives/OnlyLibs.jar on Win32 Systems

- **Use the following command to create the `OnlyLibs.jar` file containing `jmxlibid` library on the Win32 System.**

  jdk\bin\jar cf *res-dest*\archives\OnlyLibs.jar jmxlibid.dll

---

Note – Run this command from the directory containing the `jmxlibid.dll` file. Before running the command, make your current directory the directory containing the `jmxlibid.dll` file.

5.14.2 Configuration

No special requirements.

5.14.3 Execution

No special requirements.

5.14.4 Configuration Tab Name-Value Pairs

The Configuration tab name-value pairs in Table 5–28 are provided for use with the JavaTest harness test view and Test Environment dialog box when troubleshooting problems with running tests. You can view the configuration values that were used to run a specific test by choosing the Configuration tab in the Test Manager.

For a description of the test view tabs and Test Environment dialog box, see the *JavaTest Harness User's Guide: Graphical User Interface* included with this release or the JavaTest harness online help.

If you intend to set a configuration value in the command line to run tests, see Chapter 4 for a description of how to obtain the required question-tag names for your configuration.

Table 5–28 lists the name-values pairs displayed in the Configuration tab for the JMX API tests for advanced dynamic loading.

TABLE 5–28 JMX API Loading Test Configuration Tab Name-Value Pairs

Name	Value Description
`platform.nativeCodeSupported`	If your system supports native code, this value is `true`. This value signifies that the VM under test provides support for loading native code. If your system does not provide support for loading native code, this value must be `false`.
`resource.JMXLibPath`	A path to the directory containing HTML and JAR files needed to run JMX API tests. This is the directory *res-dest* containing the resources copied in "Copying Required HTML and JAR Files" on page 147.
`PATH` `LD_LIBRARY_PATH`	The configuration file (`.jti` file), includes all platform-specific variables required to identify the path to the library file. Use the Configuration Editor or `EditJTI` to modify the `.jti` file to use the `PATH` variable for win32 test systems, and the `LD_LIBRARY_PATH` variable for Solaris platform test systems. If you execute your tests using the JavaTest harness agent, you must set the platform-specific system environment variables (`PATH` or `LD_LIBRARY_PATH`) before you start the agent. See *jck*`/win32/bin/javatest` for a helpful script.

5.15 JMX Remote API Tests

Table 5–29 provides the location of the JMX Remote API tests and identifies the area that is tested.

TABLE 5–29 JMX Remote API Test Information

Test Information	Description or Value
Area tested	JMX Remote API
Test URLs	`api/javax_management/remote`

5.15.1 Setup

Before you execute the tests, you must start the TNS. The TNS is located in the `bin` directory of the JDK release.

▼ Starting the TNS on the Solaris Platform

- **To start the TNS on the Solaris platform, issue the following command:**

  jdk`/bin/tnameserv`

  This starts the server listening on the default port (900). You must have root access to use this port on the Solaris platform. To use a different port, start the TNS with the `-ORBInitialPort` flag and a port number. For example, to use port 1234, issue the following command:

  ```
  jdk/bin/tnameserv -ORBInitialPort 1234
  ```

▼ Starting the TNS on a Win32 System

- **To start the TNS on Win32 systems, issue the following command:**

  ```
  jdk\bin\tnameserv
  ```

  This starts the server listening on the default port (900). To use a different port, start the TNS with the `-ORBInitialPort` flag and a port number. For example, to use port 1234, issue the following command:

  ```
  jdk\bin\tnameserv -ORBInitialPort 1234
  ```

5.15.1.1 Setting Up a JMX Remote API Distributed Test Run

Many of the JMX Remote API tests communicate with a remote host during the test run. The remote host consists of a passive agent running on one of the Sun reference VM on a system other than the one being tested.

Note – The reference VM for the remote agent must be the same version as the VM under test.

See "5.6 Distributed Tests" on page 123 for setup procedures required to run distributed tests for the JMX Remote API.

5.15.2 Configuration

A number of the JMX Remote API tests use a distributed framework for testing the functionality of the JMX Remote API. See "5.6 Distributed Tests" on page 123 for information about configuring the JavaTest harness for using a passive agent required to run distributed tests.

5.15.3 Execution

No special requirements.

5.15.4 Configuration Tab Name-Value Pairs

The Configuration tab name-value pairs in Table 5–30 are provided for use with the JavaTest harness test view and Test Environment dialog box when troubleshooting problems with running tests. You can view the configuration values that were used to run a specific test by choosing the Configuration tab in the Test Manager.

For a description of the test view tabs and Test Environment dialog box, see the *JavaTest Harness User's Guide: Graphical User Interface* included with this release or the JavaTest harness online help.

If you intend to set a configuration value in the command line to run tests, see Chapter 4 for a description of how to obtain the required question-tag names for your configuration.

Table 5–30 lists the name-values pairs displayed in the Configuration tab for the JMX Remote API tests.

TABLE 5–30 JMX Remote API Test Configuration Tab Name-Value Pairs

Name	Value Description
`ORBHost`	The name of a host or machine running the TNS. Example Configuration tab value: `discovery`
`ORBPort`	The port number listened to by the TNS. Example Configuration tab value: `5555`

`network.tcpPortRange`	Some tests exercise the system's ability to request a specific port for a TCP connection. This port is determined dynamically each time the test is run. If `network.tcpPortRange` is 0 (zero), the test uses the OS to determine a free port that can be used for the test. This is the preferred choice, and is suitable for most operating systems. However, on some systems, the set of ports that are allocated by the OS is different from the set of ports that can be specifically requested by client code. In this case, `network.tcpPortRange` indicates to the test the range of ports that are available. The range is represented as *port1-port2* where *port1* and *port2* are the bounds of the desired range, with the following limitations:
	`1<=port1<=port2<=65535`
	Example Configuration tab value:
	`2048-5096`

5.16 JNI Implementation Tests

Table 5–31 provides the location of the Java Native Interface (JNI) implementation tests and identifies the area that is tested.

TABLE 5–31 JNI API Test Information

Test Information	Description or Value
Area tested	JNI

Test URLs	`vm/jni`
	`vm/instr/invokeinterface`
	`vm/instr/invokespecial`
	`vm/instr/invokestatic`
	`vm/instr/invokevirtual`
	`vm/classfmt/atr/atrnew002/atrnew00213m1`
	`vm/classfmt/atr/atrnew002/atrnew00214m1`

5.16.1 Setup

The JCK includes tests for JNI implementation. The JNI API provides a native programming interface that enables Java class files and methods that run inside a VM to interoperate with applications and libraries written in other programming languages (such as C, C++, and assembly). The JCK provides source code for the native portion of JNI implementation tests. However, the JCK does not provide native binary code of the JNI implementation tests for a particular platform.

Before running JNI implementation tests in the JavaTest harness, the following conditions must be satisfied as described in this section:

- You must compile the `jckjni` C source code into a native dynamic library on the licensee's native platform (using a C or C++ compiler).
- Your Java platform must include an implementation of the `System.loadLibrary(String)` method.
- Your environment must be correctly set to load the `jckjni` library before tests are run (`$LD_LIBRARY_PATH` on Solaris platforms or `%PATH%` on Win32 systems).

Note – The JNI implementation tests, when calling `System.loadLibrary()`, might encounter a security constrained Security Manager. To pass the JNI implementation tests, such an implementation must provide a way to grant the JNI tests permission to load native libraries.

Each JNI implementation test attempts to load a small library. If the test cannot load the library, the test tries to load the entire `jckjni` library. These libraries are not provided with the JCK, but you can create them for development and debugging purposes.

Note – A common mistake made with JNI implementation test libraries is to attempt to use precompiled libraries for different versions of the JCK. The JNI implementation tests provide useful diagnostics for such cases. If the loaded `jckjni` library does not match the JCK, the JNI implementation test, `vm/jni/CheckLibrary/clib001/clib00101m1/clib00101m1.html`, fails with the appropriate diagnostics.

JNI implementation tests can be compiled by the ANSI C compatible compiler or by the C++ compiler. Any platform-specific usage is declared as a macro that is utilized by all of the JNI implementation tests. The JCK provides header files that define all of the macros for the reference platforms and other platforms (`md` stands for "machine dependent") in the following directory:

`jck``/src/share/lib/jni/include/`**`platform`**`/*_md.h`

Note – The `*_md.h` files can be created and ported by JCK users either for or to other platforms.

To pass all JVM tests, you must build the `jckjni` library by compiling *`jck`*`/src/share/lib/jni/jckjni.c` and placing the resultant library in the *`jck`*`/lib` directory.

Before building the `jckjni` library, you must consider the following factors:

- If a JNI implementation test library is linked dynamically, you must set up a platform-dependent system environment variable (such as `LD_LIBRARY_PATH` on the Solaris platform or `PATH` on Win32) to load the `jckjni` libraries.
- In the C source for the JNI tests, the platform-specific `jlong`, `jfloat`, `jdouble`, and `jchar` type definitions are used. Actual implementations of these types might differ. As a result, all JNI implementation tests use macros instead of direct operations on `jlong`, `jfloat`, `jdouble`, and `jchar` values. For example, the list of `jlong`, `jfloat`, `jdouble`, and `jchar` macros for the Solaris platform is defined in *`jck`*`/src/share/lib/jni/include/solaris/jckjlong_md.h`

  If you redefine any of these macros, ensure the new header files are included when the `jckjni` library is built for that platform.

- Some JNI implementation routines use the `va_list` type to deal with the variable number of arguments passed from native code to Java methods. Implementation of the `va_list` type might differ on various platforms. For that reason, specific macros for `va_list` usage are defined in *jck*`/src/share/lib/jni/include/jckjni.h`

  To meet local `va_list` implementation requirements, you might need to redefine these.
- The `jckjni.c` file contains a list of `#include` statements that refer to the test source files in the *jck*`/tests` directory.

See "Compiling `jckjni.dll` for Win32 Systems Using MSVC++" on page 159 and "Compiling `libjckjni.so` for the Solaris Platform" on page 160 for the procedures required to build the `jckjni` library on Win32 systems and the Solaris platform.

▼ Compiling `jckjni.dll` for Win32 Systems Using MSVC++

- **Use the command lines from Example 5–8 to compile the JNI implementation tests on Win32 systems using the MSVC++ compiler.**

  Substitute the exact JCK and destination paths for the strings *jck* and *dest*.

Example 5–8 MSVC++ Compile of JNI Tests for Win32 Systems

```
cl /DWIN32 /D_WINDOWS /Ijck \
/Ijck\src\share\lib\jni\include \
/Ijck\src\share\lib\jni\include\win32 \
/LD /MD /Fodest\jck\jni.obj /Fedest\jckjni.dll \
jck\src\share\lib\jni\jckjni.c
```

5.16.2 Command-Line Options for Compiling JNI Implementation Tests on Win32 Systems

Table 5–32 describes the available command-line options for compiling JNI tests on Win32 systems.

TABLE 5–32 Command-Line Options for Compiling JNI Implementation Tests on Win32 Systems

Option	Description
`cl`	MSVC++ compiler.
`/DWIN32`	Define `WIN32` for C preprocessor.

TABLE 5–32 Command-Line Options for Compiling JNI Implementation Tests on Win32 Systems *(Continued)*

Option	Description
`/D_WINDOWS`	Define `_WINDOWS` for C preprocessor.
`/I`*jck*	Add the JCK directory to the `include` search path.
`/I`*jck*`\src\share\lib\jni\include`	Add `jckjni.h` to include search path.
`/I`*jck*`\src\share\lib\include\win32`	Add `jck*_md.h` to include search path.
`/LD /MD`	Create `.DLL` and link it with `MSVCRT.LIB` (multi-threaded runtime).
`/Fe`*dest*`\jckjni.dll`	Specify the output file name. *dest* represents the path of the destination directory.
`/Fo`*dest*`\jckjni.obj`	Specify the intermediate object file name.
jck`\src\share\lib\jni\jckjni.c`	Input file.

Note – To use the MSVC++ compiler, you might need to set up environment variables, such as `INCLUDE`, `LIB`, and `PATH`. You might use file `vsargs32.bat` from MSVC++ to do this.

▼ Compiling `libjckjni.so` for the Solaris Platform

- **Use the command lines from Example 5–9 to compile the JNI implementation tests for the Solaris platform.**

  Substitute the exact JCK and destination paths for the strings *jck* and *dest*.

Example 5–9 C Compile of JNI Implementation Tests for the Solaris Platform

```
cc -G -KPIC -o dest/libjckjni.so -Ijck \
-Ijck/src/share/lib/jni/include \
-Ijck/src/share/lib/jni/include/solaris \
jck/src/share/lib/jni/jckjni.c
```

5.16.3 Command-Line Options for Compiling JNI Implementation Tests on the Solaris Platform

Table 5–33 describes the available command-line options for compiling the JNI implementation tests on the Solaris platform.

TABLE 5–33 Command-Line Options for Compiling JNI Implementation Tests on the Solaris Platform

Option	Description
`cc`	C Compiler.
`-G`	Generate a shared library.
`-KPIC`	Generate position-independent code.
`-o` *dest*`/libjckjni.so`	Specify the output file name. *dest* represents the path of the destination directory.
`-I`*jck*	Add the JCK directory to the `include` search path.
`-I`*jck*`/src/share/lib/jni/include`	Add `jckjni.h` to the `include` search path.
`-I`*jck*`/src/share/lib/jni/include/solaris`	Add a directory containing platform- dependent `jck*_md.h` files to the include search.
jck`/src/share/lib/jni/jckjni.c`	Input file.

▼ Compiling `libjckjni.so` for the Solaris Platform Using Gnu C

- **Use the command lines from Example 5–10 to build the library by using the Gnu C compiler for the Solaris platform.**

  Substitute the exact JCK and destination paths for the strings *jck* and *dest*.

Example 5–10 Gnu C Compile of JNI Tests for the Solaris Platform

```
gcc -fPIC -shared -o dest/libjckjni.so -Ijck \
-Ijck/src/share/lib/jni/include \
-Ijck/src/share/lib/jni/include/solaris \
jck/src/share/lib/jni/jckjni.c
```

5.16.4 Command-Line Options for Compiling JNI Implementation Tests Using Gnu C

Table 5–34 describes the available Gnu C command-line options in detail.

TABLE 5–34 Command-Line Options for Compiling JNI Implementation Tests Using Gnu C

Option	Description
`gcc`	Gnu C Compiler.
`-fPIC`	Generate position-independent code, suitable for dynamic linking.
`-shared`	Generate a shared object.
`-o` *dest*`/libjckjni.so`	Specify the output file name. *dest* represents the path of the destination directory.
`-I`*jck*	Add the JCK directory to the `include` search path.
`-I`*jck*`/src/share/lib/jni/include`	Add `jckjni.h` to the `include` search path.
`-I`*jck*`/src/share/lib/jni/include/solaris`	Add a directory containing platform-dependent `jck*_md.h` files to the `include` search path.
jck`/src/share/lib/jni/jckjni.c`	Input file name and location.

5.16.5 Configuration

No special requirements.

5.16.6 Execution

No special requirements.

5.16.7 Configuration Tab Name-Value Pairs

The Configuration tab name-value pairs in Table 5–35 are provided for use with the JavaTest harness test view and Test Environment dialog box when troubleshooting problems with running tests. You can view the configuration values that were used to run a specific test by choosing the Configuration tab in the Test Manager.

For a description of the test view tabs and Test Environment dialog box, see the *JavaTest Harness User's Guide: Graphical User Interface* included with this release or the JavaTest harness online help.

If you intend to set a configuration value in the command line to run tests, see Chapter 4 for a description of how to obtain the required question-tag names for your configuration.

Table 5–35 lists the name-values pairs displayed in the Configuration tab for the JNI implementation tests.

TABLE 5–35 JNI Implementation Test Configuration Tab Name-ValuePairs

Name	Value Description
`platform.nativeCodeSupported`	If your system supports native code, this value is `true`. This value signifies that the VM under test provides support for loading native code. If your system does not provide support for loading native code, this value must be `false`.

`PATH`
 	
`LD_LIBRARY_PATH`
	The `.jti` file includes all platform-specific variables required to identify the path to the library file. Use the Configuration Editor to modify the `.jti` file to use the `PATH` variable for Win32 test systems, and the `LD_LIBRARY_PATH` variable for Solaris platform test systems.
	
	If you execute your tests using the JavaTest harness agent, you must set the platform-specific system environment variables (`PATH` or `LD_LIBRARY_PATH`) before you start the agent. See *jck*`/win32/bin/javatest` for a helpful script.

5.17 JVM TI Tests

The Java Virtual Machine Tool Interface (JVM TI) is a programming interface used by development and monitoring tools that provides both a way to inspect the state and to control the execution of applications running in the VM.

The JVM TI agent may be started at VM startup by specifying the native agent library name using a command line option. Implementations may also support a mechanism to start native agents some time after the VM has started (live phase) without specifying agents through Java platform command-line options. The details for how this is supported, is implementation specific. For example, a tool may use some platform specific mechanism, or implementation specific API, to attach to the running VM, and request it start a given native agent.

To test both types of starting JVM TI agents, the JCK provides JVM TI tests and JVM TI Live Phase tests.

Table 5–36 provides the location of the JVM TI tests and identifies the area that is tested.

TABLE 5–36 JVM TI Test Information

Test Information	Description or Value
API tested	JVM TI
Test URLs	`vm/jvmti`

5.17.1 Setup

The JVM TI tests and JVM TI Live Phase tests, like JNI implementation tests, use C code. It is not possible to define the compilation of either set of tests in a platform-independent way. Therefore, you must compile the JVM TI tests before running them within the JavaTest harness. Compile these files into a library named `jckjvmti` for the `System.loadLibrary("jckjvmti")` method to load them on your platform.

However, unlike JNI implementation tests, the JVM TI agent library is not loaded using `System.loadLibrary()` call. The JVM TI agent library is loaded by the VM and initialized during the VM startup phase. JVM TI tests and JVM TI live phase tests do `System.loadLibrary()` calls to bind native methods. See "5.16 JNI Implementation Tests" on page 157 for detailed information about JNI implementation tests.

To pass all JVM TI tests or JVM TI Live Phase tests, you must build the library `jckjvmti`. To build the library, compile `jck/src/share/lib/jvmti/jckjvmti.c` and place the resulting library in a directory that is accessible during a test run. You must specify this directory in each interview as the JVM TI library location.

The `jckjvmti.c` file contains a list of `#include` statements that refer to the test source files in the *jck*`/tests` directory.

See "Compiling `jckjvmti.dll` for Win32 Systems" on page 165 for the procedure required to build the `jckjvmti` library on Win32 systems.

See "Compiling `libjckjvmti.so` for the Solaris Platform" on page 166 and "Compiling `libjckjvmti.so` for the Solaris Platform Using Gnu C" on page 167 for the procedures required to build the `jckjvmti` library on Solaris platforms.

If your implementation supports a mechanism to allow starting native agents some time after the VM starts (live phase) then you must also set up the test system before you can run JVM TI Live Phase tests. See "5.20.1.1 Setting up a Test System to Run Java PLIS Tests " on page 181.

5.17.1.1 Setting up a Test System to Run JVM TI Live Phase Tests

The mechanism that starts JVM TI agents after the VM has started without specifying agents through Java platform command-line options is implementation specific. You must provide a wrapper class for JCK to attach to a running VM and initiate loading the native agent. The JCK uses the wrapper class to execute the JVM TI Live Phase tests. The wrapper class provided must be a subclass of the `javasoft.sqe.jck.lib.attach.AttachConnector` abstract class.

The JCK provides a `javasoft.sqe.jck.lib.attach.JVMTIAttachConnector` wrapper class to execute JVM TI Live Phase tests on JDK software.

All required Java source files of classes of the `javasoft.sqe.jck.lib.attach` package are located in the following directory:

`jck``/src/share/classes/javasoft/sqe/jck/lib/attach/`

If you provide your own wrapper class, you must use either Sun's reference Java compiler or another certified compiler to compile it on one of the reference platforms.

An example compilation command for a provided wrapper class (using JDK software on the Solaris platform) is as follows:

```
jdk/bin/javac -d . -classpath jck/classes YourAttachConnectorClass.java
```

Compiled class files of `javasoft.sqe.jck.lib.attach` are located in a corresponding subdirectory of the *jck*`/classes` directory.

▼ Compiling `jckjvmti.dll` for Win32 Systems

- **Use the command lines from Example 5–11 to compile JVM TI tests for Win32 systems using the MSVC++ compiler.**

  Substitute the exact JCK and destination paths for the strings *jck* and *dest*.

Example 5–11 MSVC++ Compile of JVM TI Tests for Win32 Systems

```
cl /DWIN32 /D_WINDOWS /Ijck
/Ijck\src\share\lib\jvmti\include
/Ijck\src\share\lib\jni\include
/Ijck\src\share\lib\jni\include\win32
/LD /MD /Fodest\jckjvmti.obj /Fedest\jckjvmti.dll
jck\src\share\lib\jvmti\jckjvmti.c
```

5.17.1.2 Command-Line Options for Compiling JVM TI Tests on Win32 Systems

Table 5–37 describes the available command-line options for compiling JVM TI tests on Win32 systems.

TABLE 5–37 Command-Line Options for Compiling JVM TI Tests on Win32 Systems

Option	Description
`cl`	MSVC++ compiler.
`/DWIN32`	Define `WIN32` for C preprocessor.
`/D_WINDOWS`	Define `_WINDOWS` for C preprocessor.
`/I`*jck*	Add the JCK directory to the `include` search path.
`/I`*jck*`\src\share\lib\jvmti\include`	Add `jckjvmti.h` to the `include` search path.
`/I`*jck*`\src\share\lib\jni\include`	Add `jckjni.h` to the `include` search path.

TABLE 5–37 Command-Line Options for Compiling JVM TI Tests on Win32 Systems *(Continued)*

Option	Description
`/I`*jck*`\src\share\lib\jni\include\win32`	Add `jck*_md.h` to the `include` search path.
`/LD /MD`	Create `.DLL` and link it with `MSVCRT.LIB` (multi-threaded runtime).
`/Fe`*dest*`\jckjvmti.dll`	Specify the output file name. The destination directory *dest* is usually *jck*`\lib`.
`/Fo`*dest*`\jckjvmti.obj`	Specify the intermediate object file name.
jck`\src\share\lib\jvmti\jckjvmti.c`	Input file name and location.

Note – To use the MSVC++ compiler, you might need to set up environment variables, such as `INCLUDE`, `LIB`, and `PATH`. You can use file `vsargs32.bat` from MSVC++ to do this.

▼ Compiling `libjckjvmti.so` for the Solaris Platform

- **Use the command lines from Example 5–12 to compile the JVM TI tests for the Solaris platform.**
  Substitute the exact JCK and destination paths for the strings *jck* and *dest*.

Example 5–12 C Compile of JVM TI Tests for the Solaris Platform

```
cc -G -KPIC -o dest/libjckjvmti.so -Ijck
-Ijck/src/share/lib/jvmti/include
-Ijck/src/share/lib/jni/include
-Ijck/src/share/lib/jni/include/solaris
jck/src/share/lib/jvmti/jckjvmti.c
```

5.17.1.3 Command-Line Options for Compiling JVM TI Tests on the Solaris Platform

Table 5–38 describes the available command-line options for compiling JVM TI tests on the Solaris platform.

TABLE 5–38 Command-Line Options for Compiling JVM TI Tests on the Solaris Platform

Option	Description
`cc`	C compiler.
`-G`	Generate a shared library.

TABLE 5–38 Command-Line Options for Compiling JVM TI Tests on the Solaris Platform *(Continued)*

Option	Description
`-KPIC`	Generate position independent code.
`-o` *dest*`/libjckjvmti.so`	Specify the output file name. The destination directory *dest* is usually *jck*`/lib`.
`-I`*jck*	Add the JCK directory to the `include` search path.
`-I`*jck*`/src/share/lib/jvmti/include`	Add `jckjvmti.h` to the `include` search path.
`-I`*jck*`/src/share/lib/jni/include`	Add `jckjni.h` to the `include` search path.
`-I`*jck*`/src/share/lib/jni/include/solaris`	Add a directory containing platform dependent `jck*_md.h` files to the include search path.
jck`/src/share/lib/jvmti/jckjvmti.c`	Input file name and location.

▼ Compiling `libjckjvmti.so` for the Solaris Platform Using Gnu C

- **Use the command lines from Example 5–13 to build the library by using the Gnu C compiler for the Solaris platform.**

  Substitute the exact JCK and destination paths for the strings *jck* and *dest*.

Example 5–13 Gnu C Compile of JVM TI Tests for the Solaris Platform

```
gcc -fPIC -shared -o dest/libjckjvmti.so -Ijck
-Ijck/src/share/lib/jvmti/include
-Ijck/src/share/lib/jni/include
-Ijck/src/share/lib/jni/include/solaris
jck/src/share/lib/jvmti/jckjvmti.c
```

5.17.1.4 Available Gnu C Command-Line Options

Table 5–39 describes the available Gnu C command-line options in detail.

TABLE 5–39 Available Gnu C Command-Line Options

Option	Description
`gcc`	Gnu C Compiler.
`-fPIC`	Generate position independent code, suitable for dynamic linking.
`-shared`	Generate a shared object.

TABLE 5–39 Available Gnu C Command-Line Options *(Continued)*

Option	Description
`-o` *dest*`/lib/jckjvmti.so`	Specify the output file name. The destination directory *dest* is usually *jck*`/lib`.
`-I`*jck*	Add the JCK directory to the `include` search path.
`-I`*jck*`/src/share/lib/jvmti/include`	Add `jckjvmti.h` to the `include` search path.
`-I`*jck*`/src/share/lib/jni/include`	Add `jckjni.h` to the `include` search path.
`-I`*jck*`/src/share/lib/jni/include/solaris`	Add a directory containing platform dependent `jck*_md.h` files to the `include` search path.
jck`/src/share/lib/jvmti/jckjvmti.c`	Input file name and location.

5.17.2 Configuration

Additional configuration is not required if your implementation does not support a mechanism that enables starting JVM TI agents after the VM has started.

If your implementation supports a mechanism that enables starting native agents after the VM starts (live phase) then you must configure the test system before you can run JVM TI Live Phase tests. To create a configuration for running the JVM TI Live Phase tests, you must specify the wrapper class name in the interview.

5.17.3 Execution

The following sections provide information about executing JVM TI tests and JVM TI Live Phase tests.

5.17.3.1 Execution of JVM TI Tests

JVM TI tests require that the JVM TI agent library is loaded by the VM before executing a test. This library might be specified by using your platform-specific options. For example, for JDK software the library might be specified by using the VM command-line options: `-agentlib` or `-agentpath`.

Execute JVM TI command-line tests separately from other JCK tests (including JVM TI Live Phase tests). In multi-VM mode, no additional settings are required. The JavaTest harness passes JVM TI-specific options to the tested VM when there is a `jvmti` keyword in the test description.

However, in single-VM mode, execute only JVM TI tests when testing a JVM TI implementation. In single-VM TI mode, use the `!jvmti&!jvmtilivephase` keyword selection

criteria to run all tests except the JVM TI JCK tests. Use `jvmti&!jvmtilivephase` keyword selection criteria in addition to other keywords to run only the JVM TI JCK tests but not JVM TI Live Phase JCK tests.

In single-VM mode, the JVM TI agent runs within the agent's VM. Start this VM with a JVM TI-specific command-line argument specifying the agent library (`jckjvmti`) to load. The argument must contain the agent option same to inform the agent of the single-VM mode. For example, for JDK this argument is the following:

```
-agentlib:jckjvmti=same
```

5.17.3.2 Execution of JVM TI Live Phase Tests

When the JVM TI Live Phase test is executed in multi-VM mode, no additional settings are required. The JavaTest harness automatically specifies the agents used in testing.

If you run the JVM TI Live Phase tests in single-VM mode, do not specify the agent library (`jckjvmti`) to load in command-line options.

JVM TI Live Phase tests cannot be executed in the same VM with other JCK tests (including JVM TI tests). In single-VM mode, use `jvmtilivephase` keyword selection criteria in addition to other keywords to run only the JVM TI Live Phase JCK tests but not JVM TI JCK tests. Use the `!jvmtilivephase` keyword selection criteria to run all but the JVM TI JCK Live Phase tests.

5.17.4 Configuration Tab Name-Value Pairs

The Configuration tab name-value pairs in Table 5–40 and Table 5–41 are provided for use with the JavaTest harness test views and Test Environment dialog box when troubleshooting problems with running tests. You can view the configuration values that were used to run a specific test by choosing the Configuration tab in the Test Manager.

For a description of the test view tabs and Test Environment dialog box, see the *JavaTest Harness User's Guide: Graphical User Interface* included with this release or the JavaTest harness online help.

If you intend to set a configuration value in the command line to run tests, see Chapter 4 for a description of how to obtain the required question-tag names for your configuration.

Table 5–40 lists the name-values pairs displayed in the Configuration tab for the JVM TI tests.

TABLE 5–40 JVM TI Test Configuration Tab Name-Value Pairs

Name	Value Description
`platform.jvmtiSupported`	If your system supports JVM TI, this value is `true`. This value signifies that the JVM software implementation under test provides support for JVM TI. If your system does not provide support for JVM TI, this value must be `false`.
`PATH` `LD_LIBRARY_PATH`	The `.jti` file includes all platform-specific variables required to identify the path to the library file. Use the Configuration Editor to modify the `.jti` file to use the `PATH` variable for Win32 test systems, and the `LD_LIBRARY_PATH` variable for Solaris platform test systems. If you execute your tests using the JavaTest harness agent, you must set the platform-specific system environment variables (`PATH` or `LD_LIBRARY_PATH`) before you start the agent. See *jck*`/win32/bin/javatest` for a helpful script.

Table 5–41 lists the name-values pairs displayed in the Configuration tab for the JVM TI Live Phase tests.

TABLE 5–41 JVM TI Live Phase Test Configuration Tab Name-Value Pairs

Name	Value Description
`platform.jvmtiLivePhaseSupported`	String value that indicates whether a JVM TI| Live Phase implementation is provided for testing. If a JVM TI Live Phase implementation is provided, this value must be `yes`.
`$jvmtiLivePhaseLauncherImpl`	String value that indicates wrapper class name. For example, for Sun's JDK specify this value: `javasoft.sqe.jck.lib.attach.VMTIAttachConnector`
`PATH` `LD_LIBRARY_PATH`	The `.jti` file includes all platform-specific variables required to identify the path to the library file. Use the Configuration Editor to modify the `.jti` file to use the `PATH` variable for Win32 test systems, and the `LD_LIBRARY_PATH` variable for Solaris platform test systems. If you execute your tests using the JavaTest harness agent, you must set the platform-specific system environment variables (`PATH` or `LD_LIBRARY_PATH`) before you start the agent. See *jck*`/win32/bin/javatest` for a helpful script.

5.18 Java Authentication and Authorization Service Tests

Table 5–42 provides the location of the Java Authentication and Authorization Service (JAAS) tests and identifies the areas that are tested.

TABLE 5–42 JAAS Test Information

Test Information	Description or Value
Areas tested	`javax.security`
Test URLs	`api/javax_security`

5.18.1 Setup

No special requirements.

5.18.2 Configuration

The authorization and the login configuration policy files must be configured to run the JAAS tests. You can set them from either the JavaTest harness GUI or the command-line interface.

5.18.2.1 Authorization Policy File

If you do not set the authorization policy file in the JCK configuration file, you must use the `-D` switch to set it in the command line shown in the following example:

```
-Djava.security.auth.policy==jck/lib/jck.auth.policy
```

Note – The == symbol denotes absolute assignment. This causes the values in this file to override any previous settings.

5.18.2.2 Login Configuration Policy File

If you do not set the login configuration policy file in the JCKF configuration file, you must use the `-D` switch to set it in the command line shown in the following example:

```
-Djava.security.auth.login.config==jck/lib/jck.auth.login.config
```

Note – The == symbol denotes absolute assignment. This causes the values in this file to override any previous settings.

5.18.2.3 Specifying Policy Files Statically

Some testing environments (for example, the JavaPlug-In software) require the security files to be specified statically. You must edit the *jdk*`/jre/lib/security/java.security` file to point to the relevant security policy files. The following sections describe how to make those edits.

Java Platform Policy File

The location of policy configurations can be statically set by specifying their respective URLs in the `policy.url.`*n* property, where *n* is a consecutively numbered integer starting with 1.

Example:

```
policy.url.1=file:///jck/lib/jck.policy
```

If multiple configurations are specified (*n* >= 2), they are read and a union of the configurations is created.

If the location of the configuration is not set in *jdk*`/jre/lib/security/java.security`, and is not specified dynamically on the command line, the runtime attempts to load a default policy from the following file:

```
${user.home}/.java.policy
```

Login Configuration File

The location of login configurations can be statically set by specifying their respective URLs in the `login.config.url.`*n* property, where *n* is a consecutively numbered integer starting with 1.

For example:

```
login.config.url.1=file:///jck/lib/jck.auth.login.config
```

If multiple configurations are specified (*n* >= 2), they are read and a union of the configurations is created.

If the location of the configuration is not set in *jdk*`/jre/lib/security/java.security`, and is not specified dynamically on the command line, the Java Authorizing Authentication Service attempts to load a default configuration from the following file:

```
${user.home}/.java.login.config
```

Authorization Policy File

The location of the access control policies can be statically set by specifying their respective URLs in the `auth.config.url.`*n* property, where *n* is a consecutively numbered integer starting with 1.

For example:

```
auth.policy.url.1=file:///jck/lib/jck.auth.policy
```

If multiple configurations are specified (*n* >= 2), they are read and a union of the configurations is created.

If the location of the policy is not set in *jdk*`/jre/lib/security/java.security`, and is not specified dynamically on the command line, the JAAS access control policy defaults to an empty policy.

5.18.3 Execution

No special requirements.

5.18.4 Configuration Tab Name-Value Pairs

The Configuration tab name-value pairs in Table 5–43 are provided for use with the JavaTest harness test view and Test Environment dialog box when troubleshooting problems with running tests. You can view the configuration values that were used to run a specific test by choosing the Configuration tab in the Test Manager.

For a description of the test view tabs and Test Environment dialog box, see the *JavaTest Harness User's Guide: Graphical User Interface* included with this release or the JavaTest harness online help.

If you intend to set a configuration value in the command line to run tests, see Chapter 4 for a description of how to obtain the required question-tag names for your configuration.

Table 5–43 lists the name-values pairs used in the configuration and displayed in the Configuration tab that set the source of the security authentication policy file and the security authorization login file.

TABLE 5–43 Java Security Authorization Policy and Login Test Configuration Tab Name-Value Pairs

Name	Value Description

`platform.JavaSecurityAuthPolicy`	A string value that indicates whether the system under test uses a system property or other means to set the Java Security Authentication Policy.
	
	For the VM, a system property is used to set the Java Security Authentication Policy. The string must be "`java.security.auth.policy`". JCK test programs can then execute instructions similar to the following:
	
	`System.setProperty\`
	`("java.security.auth.policy",\`
	`"${jck_directory}/lib/\`
	`jck.auth.policy");`
	
	Example Configuration tab value:
	
	"`java.security.auth.policy`"
	
	If the VM uses other means to source the authentication policy file (such as a URL in the `$jdk/jre/lib/security/java.security` file or an environment variable), the value must be "`none`". The JCK test programs do not execute `System.setProperty(String,String)` instructions when the value is "`none`".
	
	Example Configuration tab value:
	
	"`none`"

TABLE 5–43 Java Security Authorization Policy and Login Test Configuration Tab Name-Value Pairs *(Continued)*

Name	Value Description

`platform.JavaSecurityAuthLoginConfig`	A string value that indicates whether the system under test uses a system property or other means to set the Java Security Authentication Login Policy.
	
	For the VM, a system property is used to set the Java Security Authentication Login Policy. The string is:
	
	`System.setProperty \`
	`("java.security.auth.login.\`
	`config", \`
	`"${jck_directory}/lib/\`
	`jck.auth.login.config");`
	
	Example Configuration Tab Value:
	
	`"java.security.auth.login.config"`
	
	If the VM uses other means to source the *jck*`/lib/jck.auth.login.config` authentication policy file (such as a URL in the `$jdk/jre/lib/security/java.security` file, an existing `java.login.config` file in the user home directory or an environment variable), the value must be `"none"`. JCK test programs do not execute `System.setProperty(String,String)` instructions when the value is `"none"`.
	
	Example Configuration tab value:
	
	`"none"`

5.19 Java Generic Security Service API Tests

Table 5–44 provides the location of the Java Generic Security Service API (Java GSS) tests and identifies the area that is tested.

TABLE 5–44 Java GSS Test Information

Test Information	Description or Value
API tested	`org.ietf.jgss`
Test URL	`api/org_ietf/jgss`

5.19.1 Setup

The Kerberos Key Distribution Center (KDC) must be set up either on the same machine as the system under test or on a remote machine networked to the system under test. See Appendix C for detailed information about obtaining Kerberos software and an example of setting up a KDC.

5.19.2 Configuration

If you do not set the KDC host name and realm values in the JCK configuration, you must set it statically in the command line.

5.19.2.1 Specifying the KDC Host Name and Realm Statically

If you did not set the KDC host name and realm values in the JCK configuration file, use the `-D` switch to set it in the command line (your product might use a different method):

```
-Djava.security.krb5.kdc=KDC-Machine-Host-Name \
-Djava.security.krb5.realm=KDC_Realm
```

5.19.3 Execution

No special requirements.

5.19.4 Configuration Tab Name-Value Pairs

The Configuration tab name-value pairs in Table 5–45 are provided for use with the JavaTest harness test view and Test Environment dialog box when troubleshooting problems with running tests. You can view the configuration values that were used to run a specific test by choosing the Configuration tab in the Test Manager.

For a description of the test view tabs and Test Environment dialog box, see the *JavaTest Harness User's Guide: Graphical User Interface* included with this release or the JavaTest harness online help.

If you intend to set a configuration value in the command line to run tests, see Chapter 4 for a description of how to obtain the required question-tag names for your configuration.

Table 5–45 lists the name-values pairs displayed in the Configuration tab for the Java GSS tests.

TABLE 5–45 Java GSS Test Configuration Tab Name-Value Pairs

Name	Value Description

`platform.Krb5LoginModuleConfigured`	This value indicates that the login module required to run Java GSS tests is configured in the JRE software.
	
	The login module acts as a gateway to the Kerberos KDC and enables `org.iet.jgss` applications to log in into a KDC. See Appendix C for a description of the KDC.
	
	The VM provides a login module, `com.sun.security.auth.module.Krb5LoginModule`, that can be adopted by other implementations. However, the Kerberos login module might also be bundled in a different JAR file on different platforms. Perform a query command to identify the JAR file containing the Kerberos login module.
	
	Example query command:
	
	`%jar -tf ${JDK}/jre/lib/rt.jar | \`
	`grep Krb5LoginModule \`
	`com/sun/security/auth/module/\`
	`Krb5LoginModule.class`
	
	If the login module is bundled with the runtime JAR file and this value is true, then the Java GSS tests are run (provided the remaining variables in this category have valid values).
	
	Example Configuration tab value:
	
	`true`
	
	If this value is `false`, the Java GSS tests are not run.

TABLE 5–45 Java GSS Test Configuration Tab Name-Value Pairs *(Continued)*

Name	Value Description

`platform.JavaSecurityKrb5KDC`	A string value that indicates whether the system under test uses a Java technology system property or other means to set the Java Security Kerberos Version 5 Key Data Center.
	
	Under normal circumstances, this string is `"java.security.krb5.kdc"`.
	
	JCK test programs can then execute instructions similar to the following:
	
	`System.setProperty \`
	`("java.security.krb5.kdc", \`
	`"KDC-Machine-Host-Name");`
	
	Example Configuration tab value:
	
	`"java.security.krb5.kdc"`
	
	If the VM uses some other means to set the `KDC-Machine-Host-Name`, then this value must be `"none"`. The JCK test programs do not execute the `System.setProperty(String, String)` instruction when the value is `"none"`.
	
	Example Configuration tab value:
	
	`"none"`

`platform.KDCHostName`	The string value of the machine name or host name of the KDC. The JCK test programs use this variable in conjunction with `platform.JavaSecurityKrb5KDC` to set the Java platform system property:
	
	`System.setProperty \`
	`("java.security.krb5.kdc", \`
	`"KDC-Machine-Host-Name");`
	
	Warning: The KDC machine's clock cannot vary more than 5 minutes from the test machine clock.

TABLE 5–45 Java GSS Test Configuration Tab Name-Value Pairs *(Continued)*

Name	Value Description

`platform.JavaSecurityKrb5Realm`	A string value that indicates whether the system under test uses a Java technology system property or other means to set the Java Security Kerberos Version 5 Realm.
	
	Under normal circumstances, this string is `"java.security.krb5.realm"`.
	
	The JCK test programs can then execute instructions similar to:
	
	`System.setProperty \`
	`("java.security.krb5.realm", \`
	`"KDC_Realm");`
	
	Example Configuration tab value:
	
	`"java.security.krb5.realm"`
	
	If the VM uses some other means to set the `KDC_Realm`, this value must be `"none"`.
	
	The JCK test programs do not execute the `System.setProperty(String, String)` instruction if the value is `"none"`.
	
	Example Configuration tab value:
	
	`"none"`

`platform.KDCRealm`	The string value of the KDC realm. The realm is a domain parameter that Kerberos uses to group users. The JCK test programs use this variable in conjunction with `platform.JavaSecurityKrb5Realm` to set the Java platform system property:
	
	`System.setProperty \`
	`("java.security.krb5.realm", \`
	`"KDC_Realm");`

`platform.KerberosServerUsername`	A string value of the Kerberos principal within the realm and under KDC (the user name of a login account). Example Configuration tab value: `"user1"`
`platform.KerberosServerPassword`	A string value password for the Kerberos server user name. Example Configuration tab value: `"secretPassword1"`

TABLE 5–45 Java GSS Test Configuration Tab Name-Value Pairs *(Continued)*

Name	Value Description
`platform.KerberosClientUsername`	A string value for the client user account. Example Configuration tab value: `"user2"`
`platform.KerberosClientPassword`	A string value password for the client user name. Example Configuration tab value: `"secretPassword2"`

5.20 Java Programming Language Instrumentation Services Tests

The Java Programming Language Instrumentation Services (Java PLIS) implementation in the Java programming language is represented by a set of Java class files that provide services allowing Java programming language agents to instrument programs running on the Java virtual machine. The instrumentation agents are specified through Java platform command-line options (`-javaagent:jarpath=options`) and invoked individually when a VM is launched in the order that they appear in the command line.

Implementations might also support a mechanism to start instrumentation agents some time after the VM has started (live phase) without specifying agents through Java platform command-line options. For example, an implementation might provide a mechanism that allows a tool to attach to a running application, and initiate the loading of the instrumentation agent into the running application.

To test both types of starting instrumentation agents, the JCK provides Java PLIS tests and live phase tests.

Table 5–46 provides the location of the Java PLIS tests and identifies the areas that are tested.

TABLE 5–46 Java PLIS Test Information

Test Information	Description or Value
Area tested	`java.lang.instrument.Instrumentation`
Test URLs	`api/java_lang/instrument`

5.20.1 Setup

Set up the Java PLIS tests using the mechanism supported by the implementation. Several Java PLIS tests use C code to check that when automatic native method resolution fails, it can be correctly retried using supplied prefix and enable bytecode instrumentation to be applied to native methods. Because these tests use C code, it is not possible to supply a platform-independent version of the tests. Therefore, you must compile these tests before running them. The files must be compiled into a library named `jckjni` for loading using the method `System.loadLibrary("jckjni")`.

When building the `jckjni` library, if the library is linked dynamically, you must set up a platform-dependent system environment variable such as LD_LIBRARY_PATH on the Solaris platform, or PATH on Win32 that specifies the path to the `jckjni` library.

See "Compiling `jckjni.dll` for Win32 Systems Using MSVC++" on page 159 and "Compiling `libjckjni.so` for the Solaris Platform" on page 160 for the procedures required to build the `jckjni` library on Win32 systems and the Solaris platform.

5.20.1.1 Setting up a Test System to Run Java PLIS Tests

No special requirements are required if your implementation does not support any mechanism to allow starting instrumentation agents some time after the VM has started.

If your implementation supports a mechanism to allow starting agents some time after the VM has started (live phase) then you must set up the test system before you can run live phase tests. The following procedures are required for this.

5.20.1.2 Setting up a Test System to Run Java PLIS Live Phase Tests

The mechanism to start instrumentation agents after the VM has started without specifying agents through Java platform command-line options is implementation specific. You must provide a wrapper class for JCK to attach to a running VM and initiate loading the agent. The JCK uses the agent to execute the live phase tests. The wrapper class provided must be a subclass of the `javasoft.sqe.jck.lib.attach.AttachConnector` abstract class.

The JCK provides a wrapper class `javasoft.sqe.jck.lib.attach.JPLISAttachConnector` to execute live phase tests on JDK software. All required Java source files of classes of package `javasoft.sqe.jck.lib.attach` are located in the following directory:

```
jck/src/share/classes/javasoft/sqe/jck/lib/attach/
```

If you provide your own wrapper class, you must use either Sun's reference Java compileror another certified compiler to compile it on one of the reference platforms.

An example compilation command for a provided wrapper class (using JDK software on the Solaris platform) is as follows:

```
jdk/bin/javac -d . -classpath jck/classes YourAttachConnectorClass.java
```

Compiled class files of `javasoft.sqe.jck.lib.attach` are located in a corresponding subdirectory of the *jck*`/classes` directory.

5.20.2 Configuration

Additional configuration is not required if your implementation does not support a mechanism that enables starting instrumentation agents some time after the VM has started.

If your implementation supports a mechanism that enables starting instrumentation agents after the VM has started (live phase) then you must configure the test system before you can run live phase tests. To create a configuration for running the live phase tests, you must specify the wrapper class name in the interview.

5.20.3 Execution

The JCK tests for Java PLIS is represented by the following two JAR files containing Java PLIS agents classes, corresponding manifest files, and classes used by agents for testing:

- `jck/lib/agent1.jar`
- `jck/lib/agent2.jar`

Java PLIS agents use classes from *jck*`/lib/javatest.jar` and *jck*`/lib/jtjck.jar`. In single-VM mode, *jck*`/lib/javatest.jar` and *jck*`/lib/jtjck.jar` must be in the classpath.

The following sections describe the two Java PLIS execution models supported by JCK.

5.20.3.1 Execution of Java PLIS Tests

When the Java PLIS test is executed in multi-VM mode, no additional settings are required. The JavaTest harness automatically specifies the agents used in testing.

If you run the test in single-VM mode, specify the JAR files as Java PLIS command-line options.

Java PLIS command-line tests cannot be executed in the same VM machine with other JCK tests (including live phase tests). In single-VM mode, only execute Java PLIS tests when testing a Java PLIS implementation. In single-VM mode, use the `!jplis` keyword selection criteria to run all but the Java PLIS JCK command-line tests.

Only one Java PLIS test is in JCK 6b, represented by two agents contained in two JAR files. In addition to standard keyword selection criteria, use the `jplis` keyword selection criteria to run only the Java PLIS JCK test.

Example: If the test is executed using Sun's plug-in for Netscape on a UNIX system, specify the following option in the plug-in's control panel:

```
-classpath=jck/lib/javatest.jar:jck/lib/jtjck.jar
```

As additional options in the plugin's Control Panel, specify the following:

```
-javaagent:jck/lib/agent1.jar=testsuiteRootDirURL,opt1 \
-javaagent:jck/lib/agent2.jar=testsuiteRootDirURL,opt2 \
```

Example: In single-VM mode, if the test is executed with a stand-alone JavaTest harness agent running on a UNIX system using Sun's JDK software, specify the following code in the JavaTest harness agent command line when the Java PLIS test is executed.

```
java -classpath jck/lib/javatest.jar:jck/lib/jtjck.jar:jck/classes \
-javaagent:jck/lib/agent1.jar=testsuiteRootDirURL,opt1 \
-javaagent:jck/lib/agent2.jar=testsuiteRootDirURL,opt2 \
com.sun.javatest.agent.AgentMain JavaTest agent options
```

In the example code, the following variables and options are used:

- *testsuiteRootDirURL* is a URL of the JCK test suite root directory *jck*`/tests`.

  If you run the JavaTest harness agent on the Solaris platform, *testsuiteRootDirURL* is `file://`*jck*`/tests/`. For example, on UNIX systems, `/java/jck` might be used in the command.

  If you run the JavaTest harness agent on the Windows OS, *testsuiteRootDirURL* is `file:///`*jck*`/tests/`.

  jck is the full `/` separated path to the directory where JCK is installed. For example, on Windows systems, `c:/java/JCK` might be used in the command.
- `opt1` and `opt2` are string literals without special meaning.

5.20.3.2 Execution of Java PLIS Live Phase Test

When the Java PLIS Live Phase test is executed in multi-VM mode, no additional settings are required. The JavaTest harness automatically specifies the agents used in testing.

If you run the live phase test in single-VM mode, do not specify the instrumentation JAR files `jck/lib/agent1.jar` and `jck/lib/agent2.jar` in command-line options.

The Java PLIS Live Phase test cannot be executed in the same VM with other JCK tests (including the Java PLIS test). In single-VM mode, use the `!jplislivephase` keyword selection criteria to run all but the live phase tests.

Only one Java PLIS Live Phase test is in JCK 6b, represented by two agents contained in two JAR files. In addition to standard keyword selection criteria, use the `jplislivephase` keyword selection criteria to run only the Java PLIS JCK Live Phase tests.

Example: If the test is executed using Sun's plug-in for Netscape™ software on a UNIX system, specify the following option in the plug-in's control panel:

```
-classpath=jck/lib/javatest.jar:jck/lib/jtjck.jar
```

5.20.4 Configuration Tab Name-Value Pairs

The following sections provide information about the Configuration tab name-value pairs for Java PLIS tests and Java PLIS Live Phase tests.

5.20.4.1 Configuration Tab Name-Value Pairs for Java PLIS Tests

The Configuration tab name-value pairs in Table 5–47 are provided for use with the JavaTest harness test view and Test Environment dialog box when troubleshooting problems with running tests. You can view the configuration values that were used to run a specific test by choosing the Configuration tab in the Test Manager.

For a description of the test view tabs and Test Environment dialog box, see the JavaTest Harness User's Guide: Graphical User Interface included with this release or the JavaTest harness online help. If you intend to set a configuration value in the command line to run tests, see Chapter 4 for a description of how to obtain the required question-tag names for your configuration. Table 5–47 lists the name-values pairs displayed in the Configuration tab for the Java PLIS tests.

TABLE 5–47 Java PLIS Test Configuration Tab Name-Value Pairs

Name	Value Description
PATH LD_LIBRARY_PATH	The configuration file (.jti file), includes all platform-specific variables required to identify the path to the library file. Use the Configuration Editor or EditJTI to modify the .jti file to use the PATH variable for win32 test systems, and the LD_LIBRARY_PATH variable for Solaris platform test systems. If you execute your tests using the JavaTest harness agent, you must set the platform-specific system environment variables (PATH or LD_LIBRARY_PATH) before you start the agent. See the win32/bin/javatest in the JCK installation folder for a helpful script.

5.20.4.2 Configuration Tab Name-Value Pairs for Java PLIS Live Phase Test

The Configuration tab name-value pairs in Table 5–48 are provided for use with the JavaTest harness test view and Test Environment dialog box when troubleshooting problems with running tests. You can view the configuration values that were used to run a specific test by choosing the Configuration tab in the Test Manager.

For a description of the test view tabs and Test Environment dialog box, see the *JavaTest Harness User's Guide: Graphical User Interface* included with this release or the JavaTest harness online help.

If you intend to set a configuration value in the command line to run tests, see Chapter 4 for a description of how to obtain the required question-tag names for your configuration.

Table 5–48 lists the name-values pairs displayed in the Configuration tab for the Java PLIS Live Phase tests.

TABLE 5–48 Java PLIS Live Phase Test Configuration Tab Name-Value Pairs

Name	Value Description
`platform.jplisLivePhaseSupported`	String value that indicates whether a Java PLIS Live Phase implementation is provided for testing. If a Java PLIS Live Phase implementation is provided, this value must be `yes`.
`$jplisLivePhaseLauncherImpl`	String value that indicates wrapper class name. For example, for Sun's JDK software specify this value: `javasoft.sqe.jck.lib.attach.JPLISAttachConnector`
PATH LD_LIBRARY_PATH	The configuration file (.jti file), includes all platform-specific variables required to identify the path to the library file. Use the Configuration Editor or EditJTI to modify the .jti file to use the PATH variable for win32 test systems, and the LD_LIBRARY_PATH variable for Solaris platform test systems. If you execute your tests using the JavaTest harness agent, you must set the platform-specific system environment variables (PATH or LD_LIBRARY_PATH) before you start the agent. See jck/win32/bin/javatest for a helpful script.

5.21 Java Platform Scripting API Tests

Table 5–49 provides the location of the Java platform scripting API tests and identifies the area that is tested.

TABLE 5–49 `javax.script.ScriptEngineManager` Test Information

Test Information	Description or Value
API Tested	`javax.script`
Test URLs	`api/javax_script/ScriptEngineManager`

5.21.1 Setup

Before using a browser to run scripting tests, you must add the `provider1.jar` file located at *URL-TO-JCK-ROOT*`/tests/api/javax_script/ScriptEngineManager/` to the archive attribute in the HTML code used to launch the agent.

See Example 5–14 for an example of the applet with the `archive` attribute pointing to the `provider1.jar` used to launch the agent.

EXAMPLE 5–14 Applet Used to Launch a JavaTest Agent

```
<applet
code="com.sun.javatest.agent.AgentApplet"
codebase="URL-TO-ROOT/classes"
archive="../lib/javatest.jar,
../tests/api/javax_script/ScriptEngineManager/provider1.jar"
width=600
height=600 >
</applet>
```

See Appendix B for detailed information about how to use the JavaTest harness with a browser (Netscape Navigator and JavaPlug-In software) and an agent to run JCK tests.

5.21.2 Configuration

No special requirements.

5.21.3 Execution

No special requirements.

5.21.4 Configuration Tab Name-Value Pairs

None.

5.22 Java RMI Compiler Tests

Table 5–50 provides the location of the Java Remote Method Invocation (Java RMI) compiler tests and identifies the areas that are tested.

TABLE 5–50 Java RMI Compiler Test Information

Test Information	Description or Value
API Tested	Not applicable.

TABLE 5–50 Java RMI Compiler Test Information *(Continued)*

Test Information	Description or Value
Test URLs	`api/java_rmi`

5.22.1 Setup

No special requirements.

5.22.2 Configuration

No special requirements.

5.22.3 Execution

No special requirements.

5.22.4 Configuration Tab Name-Value Pairs

Some Java RMI compiler tests exercise the classes produced by the Java RMI compiler in a reference Java RMI runtime. These tests are identified with both `runtime` and `compiler` keywords.

The Configuration tab name-value pairs in Table 5–51 are provided for use with the JavaTest harness test view and Test Environment dialog box when troubleshooting problems with running tests. You can view the configuration values that were used to run a specific test by choosing the Configuration tab in the Test Manager.

For a description of the test view tabs and Test Environment dialog box, see the *JavaTest Harness User's Guide: Graphical User Interface* included with this release or the JavaTest harness online help.

If you intend to set a configuration value in the command line to run tests, see Chapter 4 for a description of how to obtain the required question-tag names for your configuration.

Table 5–51 lists the name-values pairs displayed in the Configuration tab for the Java RMI compiler tests.

TABLE 5–51 Java RMI Compiler Test Configuration Tab Name-Value Pairs

Name	Value Description

`network.tcpPortRange`	Some Java RMI tests exercise the system's ability to request a specific port for a TCP connection. This port is determined dynamically each time the test is run.
	
	If `network.tcpPortRange` is set to 0 (zero), the test uses the operating system to determine a free port that can be used for the test. This is the preferred choice and suitable for most operating systems. However, on some systems the set of ports that are allocated by the operating system are different from the set of ports that can be specifically requested by client code.
	
	In this case, `network.tcpPortRange` must be set to indicate to test the range of ports that are available. The range must be represented as `port1-port2` where `port1` and `port2` are the bounds of the desired range, with the following limitations:
	
	`1<=port1<=port2<=65535`
	
	Example Configuration tab value:
	
	`2048-5096`

5.23 Java RMI Tests

Table 5–52 provides the location of the Java RMI tests and identifies the area that is tested.

TABLE 5–52 RMI Test Information

Test Information	Description or Value
API Tested	`java.rmi and subpackages`
Test URLs	`api/java_rmi`

5.23.1 Setup

Prior to running the Java RMI Activation tests, you must start the `rmid` daemon on your platform.

Note – If you cannot launch another VM, these tests pass by default.

▼ Starting the RMI Daemon on the Solaris Platform

- **Type the following command to start the RMI daemon:**

  ```
  jdk/bin/rmid
  ```

  The default daemon runs on port 1098. To change the port, start the RMI daemon with the specific port number as an argument. For example, to use port 39741, type the following command:

  ```
  jdk/bin/rmid -port 39741
  ```

  You must start the RMI daemon so that it does not use an exec policy file. Do not confuse the exec policy file with the security policy file. To start the RMI daemon without using an exec policy file type the following command:

  ```
  jdk/bin/rmid -J-Dsun.rmi.activation.execPolicy=none
  ```

  For more information about using the RMI Daemon, see `http://java.sun.com/javase/6/docs/technotes/tools/solaris/rmid.html`.

▼ Starting the RMI Daemon on Win32 Systems

- **Type the following command to start the RMI daemon:**

  ```
  jdk\bin\rmid
  ```

  The default daemon runs on port 1098. To change the port, start the RMI daemon with the specific port number as an argument. For example, to use port 39741, type the following command:

  ```
  jdk\bin\rmid -port 39741
  ```

  You must start the `RMI` daemon so that it does not use an exec policy file. Do not confuse the exec policy file with the security policy file. To start the RMI daemon without using an exec policy file, type the following command:

  ```
  jdk\bin\rmid -J-Dsun.rmi.activation.execPolicy=none
  ```

  For more information about using the RMI Daemon, see `http://java.sun.com/javase/6/docs/technotes/tools/windows/rmid.html`.

5.23.2 Configuration

No special requirements.

5.23.3 Execution

No special requirements.

5.23.4 Configuration Tab Name-Value Pairs

The Configuration tab name-value pairs in Table 5–53 are provided for use with the JavaTest harness test view and Test Environment dialog box when troubleshooting problems with running tests. You can view the configuration values that were used to run a specific test by choosing the Configuration tab in the Test Manager

For a description of the test view tabs and Test Environment dialog box, see the *JavaTest Harness User's Guide: Graphical User Interface* included with this release or the JavaTest harness online help.

If you intend to set a configuration value in the command line to run tests, see Chapter 4 for a description of how to obtain the required question-tag names for your configuration.

Table 5–53 lists the name-values pairs displayed in the Configuration tab for the RMI tests.

TABLE 5–53 Java RMI Test Configuration Tab Name-Value Pairs

Name	Value Description

`network.tcpPortRange`	Some tests exercise the system's ability to request a specific port for a TCP connection. This port is determined dynamically each time the test is run.
	
	If `network.tcpPortRange` is 0 (zero), the test uses the OS to determine a free port that can be used for the test.
	
	This is the preferred choice, and is suitable for most operating systems. However, on some systems the set of ports that is allocated by the OS is different from the set of ports that can be specifically requested by client code.
	
	>In this case, the values of `network.tcpPortRange` indicate to the test the range of ports that are available. The range is represented as `port1-port2` where `port1` and `port2` are the bounds of the desired range with the following limitations:
	
	`1<=port1<=port2<=65535`
	
	Example Configuration tab value:
	
	`2048-5096`

5.24 Java XML Digital Signature Tests

The JCK includes tests for Java XML Digital Signature API Specification (XMLDSig). XMLDSig is a standard Java platform API for generating and validating XML Signatures.

Table 5–54 provides the location of the XMLDSig tests and identifies the areas that are tested.

TABLE 5–54 XMLDSig Test Information

Test Information	Description or Value
API Tested	`XMLDSig`

Test URLs	`api/javax_xml/crypto`
	`api/javax_xml/crypto/dom`
	`api/javax_xml/crypto/dsig`
	`api/javax_xml/crypto/dsig/dom`
	`api/javax_xml/crypto/dsig/keyinfo`
	`api/javax_xml/crypto/dsig/spec`

5.24.1 Setup

No special setup should be required.

5.24.2 Configuration

No special requirements.

5.24.3 Execution

A Java SE technology implementation must minimally support the default `XMLSignatureFactory` mechanism type, DOM. The XMLDSig tests are designed to find and test all `XMLSignatureFactory`s in your registered providers. The required DOM `XMLSignatureFactory` is tested along with any other `XMLSignatureFactory` that is present. If a DOM based `XMLSignatureFactory` is not found, many of the XMLDSig tests will fail.

If the tests do not find your default DOM `XMLSignatureFactory`, verify that it is properly registered. See "5.24.3.1 Register Your `XMLSignatureFactory` Provider" on page 192 for a description of how to register the required DOM `XMLSignatureFactory`.

5.24.3.1 Register Your `XMLSignatureFactory` Provider

If your `XMLSignatureFactory` provider is not registered, you can register the `XMLSignatureFactory` statically or dynamically.

Statically Registering Your `XMLSignatureFactory` Provider

You can statically register your `XMLSignatureFactory` provider by editing the `security.provider` property in the `java.security` file located in the `lib/security` directory of the JDK software. The `security.provider` property set in `java.security` has the following form:

```
security.provider.n=masterClassName
```

This property declares a security provider and specifies a preference order, *n*. When a specific provider is not requested, the preference order sets the order in which providers are searched for requested algorithms. The preference order is 1-based (in which 1 is the most preferred, followed by 2, and continuing through the remaining numbered sequence).

The *masterClassName* must specify the provider's master class. The provider's documentation must specify its master class. This class is always a subclass of the `java.security.Provider` class.

For example, if the master class is `COM.acme.provider.Acme` and you want to configure Acme as your third preferred provider, add the following line to the `java.security` file:

```
security.provider.3=COM.acme.provider.Acme
```

Dynamically Registering Your `XMLSignatureFactory` Provider

To register your `XMLSignatureFactory` provider dynamically, call either the `addProvider` or the `insertProviderAt` method in the `java.security.Security` class.

Dynamic registration is not persistent and can only be accomplished by trusted programs.

5.24.4 Configuration Tab Name-Value Pairs

None.

5.25 Network Tests

Table 5–55 provides the location of the network tests and identifies the area that is tested.

TABLE 5–55 Network Test Information

Test Information	Description or Value
API Tested	`java.net` `javax.net`

Test URLs	`api/java_net`
	`api/javax_net`
	`api/java_net/distributed`
	`api/javax_net/distributed`

5.25.1 Setup

Many of the network tests communicate with a remote host during the test run. The remote host consists of a passive agent running on one of the Sun reference VMs on a system other than the one being tested.

Note – The reference VM for the remote agent must be the same version as the VM under test.

See "5.6 Distributed Tests" on page 123 for setup procedures required to run network tests.

Note – The delivery of datagrams in the network is reliable but not guaranteed. If a test fails when a passive agent is run on a remote host in a different subnet, the cause of failure could be the result of one or more dropped datagram packets, not the test or your network configuration. You might need to reconfigure your network or run the passive agent on the same subnet to deliver the datagrams reliably so that the test passes.

5.25.2 Configuration

A number of the network tests use a distributed framework for testing the network API. Refer to "5.6 Distributed Tests" on page 123 for information about configuring the JavaTest harness for using a passive agent to run Distributed Tests.

5.25.3 Execution

No special requirements.

5.25.4 Configuration Tab Name-Value Pairs

The Configuration tab name-value pairs in Table 5–56 and Table 5–57 are provided for use with the JavaTest harness test view and Test Environment dialog box when troubleshooting problems with running tests. You can view the configuration values that were used to run a specific test by choosing the Configuration tab in the Test Manager.

For a description of the test view tabs and Test Environment dialog box, see the *JavaTest Harness User's Guide: Graphical User Interface* included with this release or the JavaTest harness online help.

If you intend to set a configuration value in the command line to run tests, see Chapter 4 for a description of how to obtain the required question-tag names for your configuration.

5.25.4.1 Basic Network Test Configuration Tab Name-Value Pairs

Table 5–56 lists the basic test configuration name-value pairs required to run the network tests.

TABLE 5–56 Basic Network Test Configuration Tab Name-Value Pairs

Name	Value Description
`network.localHost`	The host name and IP address of the machine you are using for testing. The value must be the same as the one returned by the `InetAddress.getLocalHost()` call. Example Configuration tab value: *name/address*
`network.testHost1`	The host name and IP address of some computer on your network. This must be a computer on your network with only one DNS name and only one IP address. Example Configuration tab value: *name/address*
`network.testHost2`	The host name and IP address of another host computer on your network. Example Configuration tab value: *name/address*

TABLE 5–56 Basic Network Test Configuration Tab Name-Value Pairs *(Continued)*

Name	Value Description

`network.tcpPortRange`	Some tests exercise the system's ability to request a specific port for a TCP connection. This port is determined dynamically each time the test is run.
	
	If `network.tcpPortRange` is set to 0 (zero), the test uses the OS to determine a free port that can be used for the test. This is the preferred choice and is suitable for most operating systems. However, on some systems the set of ports that is allocated by the OS is different from the set of ports that can be specifically requested by client code.
	
	In this case, `network.tcpPortRange` indicates to the test the range of ports that are available. The range is represented as `port1-port2` where `port1` and `port2` are the bounds of the desired range, with the following limitations:
	
	`1<=port1<=port2<=65535`
	
	Example Configuration tab value:
	
	`2048-5096`

`network.udpPortRange`	Some tests exercise the system's ability to request a specific port for a UDP connection. This port is determined dynamically each time the test is run. If you set `network.udpPortRange` to 0 (zero), the test uses the OS to determine a free port that can be used for the test.
	
	This is the preferred choice and is suitable for most operating systems. However, on some systems the set of ports that is allocated by the OS is different from the set of ports that can be specifically requested by client code. In this case, `network.udpPortRange` must be set to indicate to the test the range of ports that are available. The range is represented as `port1-port2` where `port1` and `port2` are the bounds of the desired range, with the following limitations:
	
	`1<=port1<=port2<=65535`
	
	Example Configuration tab value:
	
	`2048-5096`

5.25.4.2 Network URL Test Configuration Tab Name-Value Pairs

The configuration name-value pairs in Table 5–57 represent absolute URLs that point to some existing resource accessible from your network. The content of the resource and protocol-related information (such as header fields) must not be dynamically generated, as is the case for a CGI-generated or servlet-generated resource. Because URLConnection tests are not intended to test index generation, the URL must not point to a directory.

The string must be in the following form:

`protocol://host[:port]/path-to-resource`

Example Configuration tab value:

```
protocol://web2.javasoft.som/index.html
```

Table 5–57 contains test configuration name-value pairs required to run URL and URLConnection tests:

TABLE 5–57 Network URL Test Configuration Tab Name-Value Pairs

Name	Value Description
`network.httpURL`	Points to an existing and accessible HTTP resource. Example Configuration tab value: `http://web2.javasoft.com/index.html`
`network.ftpURL`	Points to an existing and accessible FTP resource. Example Configuration tab value: `hftp://ftp.javasoft.com/pub/README.first`

`network.fileURL`	Points to an existing and accessible file resource on the platform under test.
	Example Configuration tab value for Win32 system:
	`file:///c:\\autoexec.bat`
	Example Configuration tab value for Solaris platform:
	`file:///etc/passwd`

`network.jckHTTPProxyHost`
`network.jckHTTPProxyPort`
	The name and port number of the HTTP proxy server to be used by JCK tests if a direct connection is not used.
	Example Configuration tab value:
	`proxy.somewhere.com`
	`3128`
	If the hosts specified in `network.httpURL` and `network.ftpURL` are directly accessible, then both proxy parameters must be `NONE`.
	If you use an HTTP proxy server for HTTP and FTP connections, these configuration values must be the name and port number of your HTTP proxy server. Your runtime implementation must use the same proxy host.

5.26 Out-of-Memory Tests

Table 5–58 provides the location of the out-of-memory tests and identifies the area that is tested.

TABLE 5–58 Out-of-Memory Test Information

Test Information	Description or Value
API Tested	Virtual Machine

Test URLs	`lang/EXCP/excp006/excp00601`
	`lang/EXEC/exec010/exec01001`
	`lang/EXEC/exec010/exec01002`
	`lang/EXEC/exec010/exec01003`
	`lang/EXPR/expr056/expr05601`
	`lang/EXPR/expr064/expr06401`
	`lang/EXPR/expr070/expr07001`
	`lang/EXPR/expr141/expr14101`

JCK includes several tests that check that `OutOfMemoryError` is thrown.

5.26.1 Setup

No special requirements.

5.26.2 Configuration

No special requirements.

5.26.3 Execution

No special requirements.

5.26.4 Configuration Tab Name-Value Pairs

The Configuration tab name-value pairs in Table 5–59 are provided for use with the JavaTest harness test view and Test Environment dialog box when troubleshooting problems with running tests. You can view the configuration values that were used to run a specific test by choosing the Configuration tab in the Test Manager.

For a description of the test view tabs and Test Environment dialog box, see the *JavaTest Harness User's Guide: Graphical User Interface* included with this release or the JavaTest harness online help.

If you intend to set a configuration value in the command line to run tests, see Chapter 4 for a description of how to obtain the required question-tag names for your configuration.

Table 5–59 lists the name-values pairs displayed in the Configuration tab for the out-of-memory tests.

TABLE 5–59 Out-of-Memory Test Configuration Tab Name-Value Pairs

Name	Value Description
`platform.expectOutOfMemory`	The value `true` (the default) denotes that `OutOfMemoryError` can be thrown. The out-of-memory test completes successfully if `OutOfMemoryError` is thrown. The value `false` denotes that the test does not wait for `OutOfMemoryError` to be thrown in order to complete successfully. For successful completion, the amount of memory specified in `platform.maxMemory` must be allocated.
`platform.maxMemory`	A positive `double` value with an optional letter denotes the amount of memory (in megabytes, kilobytes, or bytes) the test will attempt to allocate.The following letters are valid: ■ `k` or `K` indicate the value is specified in kilobytes ■ `m` or `M` indicate the value is specified in megabytes The default value is 0 (zero). The test attempts to allocate the memory of the size of `platform.maxMemory`. If `OutOfMemoryError` occurs, the test passes. Otherwise the value of `platform.expectOutOfMemory` is checked. If `platform.expectOutOfMemory` is `true`, the test passes when `OutOfMemoryError` is thrown and fails when `OutOfMemoryError` is not thrown. If `platform.expectOutOfMemory` is `false`, the test passes when the specified amount of memory is allocated. A value of 0 (zero) denotes that there are no restrictions for memory allocation. This value is valid only when `platform.expectOutOfMemory` is `true`. The test passes if `OutOfMemoryError` is thrown.

5.27 Platform-Specific Values

Table 5–60 identifies the platform-specific values used in the configuration.

TABLE 5–60 Platform-Specific Values

Platform	Value
Microsoft Windows NT/2000/XP	`SystemRoot`
Microsoft Windows 98SE	`windir`
Microsoft Windows 98SE/NT/2000/XP	`PATH`
X11-based systems	`DISPLAY`

5.27.1 Setup

No special requirements.

5.27.2 Configuration

No special requirements.

5.27.3 Execution

No special requirements.

5.27.4 Configuration Tab Name-Value Pairs

The Configuration tab name-value pairs in Table 5–61 are provided for use with the JavaTest harness test view and Test Environment dialog box when troubleshooting problems with running tests. You can view the configuration values that were used to run a specific test by choosing the Configuration tab in the Test Manager.

For a description of the test view tabs and Test Environment dialog box, see the *JavaTest Harness User's Guide: Graphical User* Interface included with this release or the JavaTest harness online help.

If you intend to set a configuration value in the command line to run tests, see Chapter 4 for a description of how to obtain the required question-tag names for your configuration.

Table 5–61 lists the name-values pairs displayed in the Configuration tab for tests run on specific platforms.

TABLE 5–61 Platform-Specific Test Configuration Tab Name-Value Pairs

Name	Value Description

`SystemRoot`
`windir`
`PATH`
	The Win32 variables `SystemRoot`, `windir`, and `PATH` must be properly set for error free test runs. Windows NT, Windows 2000, and Windows XP use the `SystemRoot` variable. Windows98SE and Windows ME use the `windir` variable. These variables are defined with the multiple VM environment `execute` command entries. For single VM environments, `PATH` must be set on the command line prior to launching the JavaTest harness. You can do this in the `autoexec.bat` file or manually via the command line with one of the following actions:
	■ Specify the values on the command line using the `-D` option.
	Example command line:
	`java -Dwindir=$windir com.sun.javatest.tool.Main`
	■ Change the definition in the `.jti` file.

`DISPLAY`	For AWT and Swing tests to run correctly, users on X11-based systems must explicitly define the `DISPLAY` value either in the Configuration Editor when creating the `.jti` file, or as a system property when starting the JavaTest harness. Example command line: `java -DDISPLAY=$DISPLAY com.sun.javatest.tool.Main`

5.28 Printing Tests

Table 5–62 provides the location of the printing tests and identifies the area that is tested.

TABLE 5–62 Printing Test Information

Test Information	Description or Value

API Tested	`java.awt.print`
	`javax.print`
	`Javax.swing.JTable`

Test URLs	`api/java_awt/interactive/PrintDialogTest.html`
	`api/java_awt/interactive/PageDialogTest.html`
	`api/javax_print`
	`api/javax_swing/interactive/JTablePrintTests.html`

5.28.1 Setup

The printer must be brought online before the tests are run. Some printing tests require user interaction. For details about running interactive tests, see "5.10 Interactive Tests" on page 134.

The printing interactive tests are located at `api/javax_print/interactive/index.html`.

5.28.2 Configuration

No special requirements.

5.28.3 Execution

No special requirements.

5.28.4 Configuration Tab Name-Value Pairs

The Configuration tab name-value pairs in Table 5–63 are provided for use with the JavaTest harness test view and Test Environment dialog box when troubleshooting problems with running tests. You can view the configuration values that were used to run a specific test by choosing the Configuration tab in the Test Manager.

For a description of the test view tabs and Test Environment dialog box, see the *JavaTest Harness User's Guide: Graphical User* included with this release or the JavaTest harness online help.

If you intend to set a configuration value in the command line to run tests, see Chapter 4 for a description of how to obtain the required question-tag names for your configuration.

Table 5–63 lists the name-values pairs displayed in the Configuration tab for the printing tests.

TABLE 5–63 Printing Test Configuration Tab Name-Value

Name	Value Description
`platform.hasPrinter`	Boolean value that indicates if a printer is connected to the system being tested. If a printer is connected to the system and is online, the value must be `true`. Otherwise, the value must be `false`.

5.29 Schema Compiler

The schema compiler compiles test files and executes the resulting class files on a Sun reference runtime implementation. See Chapter 4 for information about running the tests of the devtools test suite. Table 5–64 provides the location of the schema compiler tests.

TABLE 5–64 Schema Compiler Test Information

Test Information	Description or Value
API tested	Not applicable. Tests for the API are located in the JCK runtime test suite.
Test URLs	`xml_schema`

5.29.1 Setup

Executing a schema compiler is implementation specific. Depending on whether you use the multi-VM or the single-VM execution mode, you must provide either a wrapper script or a wrapper class for JCK to execute a schema compiler.

- If you choose to start a new instance of the product from a command line each time a new test is executed, the JCK invokes the schema compiler as a separate process for each test requiring schema compilation. The schema compiler, schema generator, and tests run in multi-VM execution mode.

  For detailed instructions required to provide and use a wrapper script to execute a schema compiler in this test environment, see "5.29.1.1 Running a Schema Compiler in Multi-VM Mode" on page 202 and "Sample Scripts for Running the Reference Schema Compiler" on page 203.
- If you choose not to start a new instance of the product from a command line each time a new test is executed, all schemas are compiled or generated and tests executed in a single process, the agent's VM. The schema compiler and the schema generator run in single-VM execution mode.

  For detailed instructions required to provide and use a wrapper class to execute a schema compiler in this test environment, see "5.29.1.2 Running a Schema Compiler in Single-VM Mode" on page 204 and "Sample Class for Running the Reference Schema Compiler" on page 204.

5.29.1.1 Running a Schema Compiler in Multi-VM Mode

In multi-VM mode, the schema compiler is invoked as a separate process for each test requiring schema compilation. The JCK invokes the wrapper script for the schema compiler using the `java.lang.Runtime.exec(String[] cmdarray, String[] envp)` method, where `cmdarray` is the following:

command [classpath-option valueC] package-option valueP outputdir-option valueD [extra-options] schema-file

Optional parameters are enclosed in square brackets ([]).

- *command* - The path to a wrapper-to-schema compiler. Specify this in the Configuration Editor.
- *classpath-option* - The name of optional classpath option. Specify this either in the Configuration Editor or by using an environment variable.
- *valueC* - The value of *classpath-option* is constructed by the JCK.
- *package-option* - The package option name. Specify this in the Configuration Editor.
- *valueP* - The value of *package-option* is constructed by the JCK.
- *outputdir-option* - The output directory option name. Specify this in the Configuration Editor.
- *valueD* - The value of *outputdir-option* is constructed by the JCK.
- *extra-options* - Additional options and their values. Specify this in the Configuration Editor.
- *schema-file* - The path constructed by the JCK to the schema to be compiled.

You can use the Configuration Editor to specify the following values for the *envp* parameter:

- *CLASSPATH* environment variable - This is an alternative to the classpath option. The Configuration Editor enables you choose between these two alternatives.
- **Additional environment variables** - As required.

Sample Scripts for Running the Reference Schema Compiler

The devtools compilers test suite provides two sample scripts that work for the reference schema compiler. You can use these scripts to develop scripts for other implementations:

- `solaris/bin/xjc.sh` (or `linux/bin/xjc.sh`) - A `ksh` script. Used to run tests with the reference schema compiler on Solaris or Linux platforms.

  This script accepts the following options:

  - `-d` *DIR-NAME* - The path to the output directory.
  - `-p` *PACKAGE-NAME* - Package name to which generated `.java` classes are to belong.
  - `-javac` *PACKAGE-NAME* - (optional) Alternate path to the Java compiler used to compile generated Java sources.

    The script also requires the following environment variables to be set:

  - *JAVA_HOME* - The path to Sun reference runtime implementation.
  - *CLASSPATH* - The classpath required for TCK tests.
- `win32/bin/xjc.bat` - Windows batch script. Used to run tests with the reference schema compiler on Windows platforms. This script is similar to the `solaris/bin/xjc.sh` script.

5.29.1.2 Running a Schema Compiler in Single-VM Mode

In single-VM mode, all invocations of a schema compiler are made from the JavaTest agent running in the same VM. A wrapper class for the schema compiler must implement the following interface:

```
package com.sun.jck.lib;
public interface XjcTool {
int compile (String[] args, PrintStream out, PrintStream err)
```

The invocation of the compile method should start schema compilation. The JCK includes the following class that is fully compatible with Sun's reference implementation and can be used for schema compilation:

`com.sun.jck.lib.Xjc`

The method uses the `com.sun.tools.xjc.Driver` class to compile a schema into `.java` sources, and then uses the Java SE 6 `com.sun.tool.javac.Main` class to compile the `.java` sources into class files:

```
compile (String [] args, java.io.PrintStream out, java.io.PrintStream err)
```

Sample Class for Running the Reference Schema Compiler

The devtools test suite provides a sample wrapper class, `src/share/classes/com/sun/jck/lib/Xjc.java`, which works for the reference schema compiler, and can be used to develop wrapper classes for other schema compiler implementations. The class provides the following method:

```
public int compile(String[] args, PrintStream out, PrintStream err)
```

The method supports the following `args`:

- `-d` *DIR-NAME* - Path to the output directory.
- `-p` *PACKAGE-NAME* - Package name to which generated Java classes should belong.
- `extra_options` - Other options to pass to reference schema compiler.

5.29.2 Configuration

In the Configuration Editor, you must choose whether or not to start a new instance of the product from a command line each time a new test is executed. See "5.29.1 Setup" on page 202 for detailed instructions required to provide and use either a wrapper script or a wrapper class to execute a schema compiler.

5.29.3 Execution

No special requirements.

5.29.4 Configuration Tab Name-Value Pairs

None.

5.30 Schema Generator

The schema generator generates schemas from test files and validates the resulting schemas on the Sun reference runtime implementation. Table 5–65 provides the location of the schema generator tests.

TABLE 5–65 Schema Generator Test Information

Test Information	Description or Value
API tested	Not applicable. Tests for the API are located in the JCK runtime test suite.
Test URLs	`java2schema`

5.30.1 Setup

Executing a schema generator is implementation-specific. Depending on whether you use the multi-VM or the single-VM mode, you must provide either a wrapper script or a wrapper class for JCK to execute a schema generator.

- If you choose to start a new instance of the product from a command line each time a new test is executed, the JCK invokes the schema generator as a separate process for each test requiring schema generation. The schema compiler, schema generator, and tests will run in multi-VM mode.

  For detailed instructions required to provide and use a wrapper script to execute a schema generator in this test environment, see "5.30.1.1 Running a Schema Generator in Multi-VM Mode" on page 206 and "Sample Scripts for Running the Reference Schema Generator" on page 206.

- If you choose not to start a new instance of the product from a command line each time a new test is executed, all schemas are compiled or generated and the tests executed in a single process, the agent's VM. The schema compiler and the schema generator run in single-VM mode.

  For detailed instructions required to provide and use a wrapper class to execute a schema generator in this test environment, see "5.30.1.2 Running a Schema Generator in Single-VM Mode" on page 207 and "Sample Class for Running the Reference Schema Generator" on page 207.

5.30.1.1 Running a Schema Generator in Multi-VM Mode

In multi-VM mode, a schema generator is invoked as a separate process for each test requiring schema generation. The JCK invokes a wrapper script for a schema generator using the `java.lang.Runtime.exec(String[] cmdarray, String[] envp)` method, where `cmdarray` consists of the following:

command [classpath-option valueC] package-option valueP outputdir-option valueD [extra-options] java-file

Optional parameters are enclosed in square brackets ([ ]).

- *command* - The path to a wrapper-to-schema generator. Specify this in the Configuration Editor.
- *classpath-option* - The name of the optional classpath option. Specify this either in the Configuration Editor or by using an environment variable.
- *valueC* - The value of *classpath_option* is constructed by the TCK.
- *package-option* - The package option name. Specify this in the Configuration Editor.
- *valueP* - The value of *package-option* is constructed by the TCK.
- *outputdir_option* - The output directory option name. Specify this in the Configuration Editor.
- *valueD* - The value of *outputdir-option* is constructed by the TCK.
- *extra-options* - Additional options and their values. Specify this in the Configuration Editor.
- *java_file* - The path constructed by the TCK to the Java class file source to be compiled into schema.

You can use the test harness configuration editor to specify the following values for the `envp` parameter:

- `CLASSPATH` environment variable - Alternative to the classpath option. The Configuration Editor enables you choose between these two alternatives.
- **Additional environment variables** - As required.

Sample Scripts for Running the Reference Schema Generator

The devtools compilers test suite provides two sample scripts that work with the reference schema generator, and can be used to develop scripts for other schema generator implementations:

- `solaris/bin/schemagen.sh` (`linux/bin/xjc.sh`) - A `ksh` script. Used to run tests with the reference schema generator on Solaris or Linux platforms.

  This script accepts the following options:

  - `JAVA-HOME` - The path to the Sun reference runtime implementation.
  - `CLASSPATH` - The classpath required for TCK tests.

- `win32/bin/schemagen.bat` - Windows batch script. Used to run tests with the reference schema generator on Windows platforms. This script is similar to the `solaris/bin/schemagen.sh` script.

5.30.1.2 Running a Schema Generator in Single-VM Mode

In single-VM mode, all invocations of a schema generator are made from the JavaTest agent running in the same VM. A wrapper class to a schema generator must implement the following interface:

```
package com.sun.jck.lib;
public interface SchemaGenTool {
int generate (String[] args, PrintStream out, PrintStream err)
```

The invocation of the generate method should start schema generation.

The JCK includes the following class that is fully compatible with Sun's reference implementation and can be used for schema generation:

```
com.sun.jck.lib.SchemaGen
```

The following method generates xml schema from `.java` class file or files:

```
generate (String [] args, java.io.PrintStream out, java.io.PrintStream err)
```

Sample Class for Running the Reference Schema Generator

The devtools compilers test suite provides a sample wrapper class, `src/share/classes/com/sun/jck/lib/SchemaGen.java`, which works for the reference schema generator and which can be used to develop wrapper classes for other schema generator implementations. The class provides the following method:

```
public int generate(String[] args, PrintStream out, PrintStream err)
```

The method supports the following `args`:

- `-d` *DIR-NAME* - Path to the output directory.
- `-p` *PACKAGE-NAME* - Package name to which generated schema should belong.
- `extra-options` - Other options to pass to reference schema generator.

5.30.2 Configuration

In the Configuration Editor, you must choose whether or not to start a new instance of the product from a command line each time a new test is executed. See "5.30.1 Setup" on page 205 for detailed instructions required to provide and use either a wrapper script or a wrapper class to execute a schema generator.

5.30.3 Execution

No special requirements.

5.30.4 Configuration Tab Name-Value Pairs

None.

5.31 Security Tests

Table 5–66 provides the location of the security tests and identifies the area that is tested.

TABLE 5–66 Security Test Information

Test Information	Description or Value
API Tested	`java.security` and subpackages `javax.security` and subpackages

Test URLs	`api/java_security`
	`api/javax_crypto`
	`api/javax_security`

5.31.1 Setup

To run these tests correctly, the security policy files on the system being tested must be configured properly. If the security policy files are not set up correctly, an error message similar to the following might be displayed:

```
Exception in thread "main" java.security.AccessControlException:
access denied (java.io.FilePermission jlog.txt write) at
java.security.AccessControlContext.checkPermission(Compiled Code) at
java.security.AccessController.checkPermission(Compiled Code)
```

Use the following sections to set up and configure the security policy files on your system:

- “5.31.1.1 Specifying the Security Policy File Dynamically” on page 209
- “5.31.1.2 Configuring Security Permission for JCK ” on page 209
- “5.31.1.3 Specifying a Different Security Provider” on page 210

5.31.1.1 Specifying the Security Policy File Dynamically

The security policy files can be specified dynamically on the command line when the tests are executed. To specify a security policy file with the JDK software, include the following code in the command-line arguments:

```
-Djava.security.policy=policy-file-name
```

For example, *policy-file-name* might be `/home/user/testfile.policy`.

The following default policy file is provided:

```
jck/lib/jck.policy
```

The policy file must grant `java.security.AllPermission` for "standard" properties that can be accessed by anyone.

The same policy file must also be used for running rmid. For example, with the JDK software:

```
jdk/bin/rmid -J-Djava.security.policy=policy-file-name
```

For information and examples on creating a policy file, see `http://java.sun.com/javase/6/docs/technotes/guides/security/PolicyFiles.html`.

5.31.1.2 Configuring Security Permission for JCK

To eliminate the dependency on policy files, JCK 6b security tests perform some operations dynamically. For example, some test permissions are set to different protection domains and classes. Using these permissions, the tests can dynamically verify the compatibility of the security model and classes in your product with respect to the Java SE platform security architecture. The tests expect that the permissions in Table 5–67 are set.

TABLE 5–67 Security Permission for JCK

Permission	Type
`createClassLoader`	`java.lang.RuntimePermission`
`Policy.getPolicy`	`java.security.SecurityPermission`
`Policy.setPolicy`	`java.security.SecurityPermission`

Example 5–15 contains a default set of properties in the reference JDK software policy file format.

EXAMPLE 5–15 Permissions Set in the Reference JDK Software Policy File

```
grant {
permission java.lang.RuntimePermission "createClassLoader";
permission java.security.SecurityPermission "java.getPolicy";
permission java.security.SecurityPermission "java.setPolicy";
}
```

Note – In some systems, policy might be hard coded and cannot be customized as described in Example 5–15. In those cases, the tests are aware of what to expect and check for the correct behavior.

If you run the tests with an agent in a single-VM environment, you should grant these permissions when you start the agent. How you grant these permissions varies depending on how you run the agent.

5.31.1.3 Specifying a Different Security Provider

The reference JDK software distributions come with their own provider of cryptography algorithms. This provider is called `sun.security.provider.Sun` and is specified as a security property in the file *jdk*`/jre/lib/security/java.security`. The property is called `security.provider.`*n* where *n* is the ordering of the providers that are installed.

Some companies ship their own security cryptography provider containing implementations of cryptography algorithms. In these cases, you must add the provider to your list of approved providers before the JCK can pick up algorithms from your provider by default. For example, in Sun's JDK software this is done statically by editing the `java.security` file in the *jdk*`/jre/lib/security` directory. For each provider this file should contain a statement of the following form:

```
security.provider.n=masterClassName
```

This declares a provider and specifies its preference order *n*. The preference order is used to search for providers of requested algorithms when a specific provider is not requested. The order is 1-based: 1 is the most preferred, followed by 2, and so on. Be sure to assign your provider as `security.provider.1` to make it the most preferred. The *masterClassName* entry must specify the fully qualified name of the provider's master class.

For example, the list of providers in your `java.security` file might look like the following:

```
# # List of providers and their preference orders: #
security.provider.1=your-provider's-qualified-pathname
security.provider.2=sun.security.provider.Sun
```

When the tests are run, the correct provider is verified for compatibility.

For information about implementing a provider, see *How To Implement a Provider for the Java Cryptography Architecture* at `http://java.sun.com/javase/6/docs/technotes/guides/security/crypto/provider.html`.

Note – Your system might utilize some other method to provide this configuration information or require other configuration files (such as library files) to be modified.

5.31.2 Configuration

Special configuration steps required to run the security tests are included in "5.31.1 Setup" on page 208.

5.31.3 Execution

No special requirements.

5.31.4 Configuration Tab Name-Value Pairs

None.

5.32 Sound Tests

Table 5–68 provides the location of the tests that test the Java platform sound APIs and identifies the area that is tested.

TABLE 5–68 Sound Test Information

Test Information	Description or Value
API Tested	`javax.sound`
Test URLs	`api/javax_sound` `api/javax_sound/interactive`

5.32.1 Setup

Some sound tests require user interaction. For details about running interactive tests, see "5.10 Interactive Tests" on page 134.

The sound interactive tests are located at `api/javax_sound/interactive/index.html`.

5.32.2 Configuration

No special requirements.

5.32.3 Execution

No special requirements.

5.32.4 Configuration Tab Name-Value Pairs

The Configuration tab name-value pairs in Table 5–69 are provided for use with the JavaTest harness test view and Test Environment dialog box when troubleshooting problems with running tests. You can view the configuration values that were used to run a specific test by choosing the Configuration tab in the Test Manager.

For a description of the test view tabs and Test Environment dialog box, see the *JavaTest Harness User's Guide: Graphical User Interface* included with this release or the JavaTest harness online help.

If you intend to set a configuration value in the command line to run tests, see Chapter 4 for a description of how to obtain the required question-tag names for your configuration.

Table 5–69 lists the name-values pairs displayed in the Configuration tab for the sound tests.

TABLE 5–69 Sound Test Configuration Tab Name-Value Pairs

Name	Value Description
`platform.canPlaySound`	A boolean value that specifies if system sound resources are available for tests to use for playing sound. Example Configuration tab value: `false` **Note –** If set to `true` some platforms, such as Solaris, require that the JavaTest harness or agent are run locally or remotely (for headless platforms) with appropriate permissions to access sound system resources.
`platform.canRecordSound`	A boolean value that specifies if system sound resources are available for tests to use for playing sound. Example Configuration tab value: `false`

TABLE 5–69 Sound Test Configuration Tab Name-Value Pairs *(Continued)*

Name	Value Description
`platform.soundURL`	The URL to a sound file available to the system. This URL is used by various sound tests for playback and conversion testing. The URL must be a sound file (`wav`, `au`, `snd`) that your system is capable of playing. Example Configuration tab value: `file:///jck/tests/api/javax_sound/sound.wav`
`platform.canPlayMidi`	A boolean value that specifies whether the system can play MIDI-encoded sound. Example Configuration tab value: `false`

5.33 Static Initialization Tests

Table 5–70 provides the location of the Static Initialization tests and identifies the area that is tested.

TABLE 5–70 Static Initialization Test Information

Test Information	Description or Value
API Tested	Virtual machine
Test URL	`vm`, `lang`

5.33.1 Setup

VM tests that test static initialization might require unloading between test runs, particularly in single VM environments. It is important that these VM tests run on a fresh VM. In these cases, back-to-back test execution runs fail without unloading the classes. If you are testing in a browser, you might need to exit and restart. Each test run must be preceded by some method to ensure a clean VM if you are re-testing in a single VM environment. See Chapter 4 for more information regarding single VM environments.

5.33.2 Configuration

No special requirements.

5.33.3 Execution

No special requirements.

5.33.4 Configuration Tab Name-Value Pairs

None.

5.34 Optional Static Signature Test

Table 5–71 provides the location of the Static Signature tests and identifies the area that is tested.

TABLE 5–71 Static Signature Test Information

Test Information	Description or Value
API Tested	API signature
Test URLs	`api/signaturetest/sigtest.basic.html` `api/signaturetest/sigtest.static.html`

The static signature test is a variant of the conventional signature test, but the static version does more checking than the conventional version. It is an optional test, and requires that your classes are available in a format described in the VM specification, Chapter 4, and placed in a directory tree, a ZIP archive, or a JAR file.

The signature test uses reflection to analyze your Java technology runtime classes and is limited by the capabilities of the reflection API. The static signature test reads your class files directly and performs more validity checks that can be accomplished by using reflection. For example, it checks for any extra classes (such asclasses that are not defined in `sigfile`) added to the tested implementation.

If your class files are organized so that they can be read by this test, run this test to further ensure that your runtime system is valid.

5.34.1 Setup

No special requirements.

5.34.2 Configuration

When you select the version of the JAX-WS technology in the Configuration Editor, make sure the selected version (JAX-WS 2.0 or JAX-WS 2.1) matches the version of JAX-WS in your implementation. If the versions do not match, the tests will not run properly.

Note – See JSR 224 on the JCP web site (`http://www.jcp.org`) for more information about JAX-WS specification.

5.34.3 Execution

No special requirements.

5.34.4 Configuration Tab Name-Value Pairs

The Configuration tab name-value pairs in Table 5–72 are provided for use with the JavaTest harness test view and Test Environment dialog box when troubleshooting problems with running tests. You can view the configuration values that were used to run a specific test by choosing the Configuration tab in the Test Manager.

For a description of the test view tabs and Test Environment dialog box, see the *JavaTest Harness User's Guide: Graphical User Interface* included with this release or the JavaTest harness online help.

If you intend to set a configuration value in the command line to run tests, see Chapter 4 for a description of how to obtain the required question-tag names for your configuration.

Table 5–72 lists the name-values pairs displayed in the Configuration tab for the static signature test.

TABLE 5–72 Static Signature Test Configuration Tab Name-Value Pairs

Name	Value Description

`platform.staticSigTestClasspath`	A list of directories and `.zip` or `.jar` files that contain the tested classes. Items in the list are separated by a system-dependent path separator. This classpath must include all class libraries that are part of the JRE software except for the classes that implement Technologies in addition to the Technology Under Test (see Chapter 2, section "2.2.2 Rules for Java SE 6 Technology Products" on page 34, compatibility rule 7.a)
	Example Configuration tab value:
	`/usr/local/java/solsparc/jre/lib/\`
	`rt.jar`

`platform.supportStaticSigTest`	Boolean value that indicates whether the static signature test must be executed on the system being tested. The static signature test must be executed if the class files for the JRE software are delivered in a directory hierarchy, or in a ZIP or JAR file. If the class files for the JRE are delivered in a directory hierarchy, or in a ZIP or JAR file, this value must be `true`. Otherwise, it must be `false`.

5.35 VM Tests

Java SE platform virtual machines use two strategies of class verification, a default of typechecking and a fallback mode of type inference. Typechecking is a new strategy of class verification used by Java SE platform. Verifying classes with typechecking is based on the class format extensions implemented for class files whose version number is 50.0 or greater. Class files whose version number is 50.0 or greater must be verified by typechecking.

Previous versions of the VM used type inference to verify class files. To enable backwards compatibility, the Java SE platform only allows support for a type inference fallback mode if the class file version number is 50.0. When the fallback mode is enabled, type inference is used to verify classes if typechecking fails. If the product VM supports the type inference fallback mode, class verification with both typechecking and type inference must be complaint with the specification.

Table 5–73 provides the location of the VM tests and identifies the area that is tested.

TABLE 5–73 VM Test Information

Test Information	Description or Value
API Tested	VM

TABLE 5–73 VM Test Information *(Continued)*

Test Information	Description or Value
Test URLs	`vm`

5.35.1 Setup

No special requirements.

5.35.2 Configuration

No special requirements.

5.35.3 Execution

When a tested VM supports the fallback mode, you must run the JCK at least twice. In one test run, each interview and the tested VM must be configured so that the fallback mode is activated. In the second test run, each interview and the tested VM must be configured so that the fallback mode is either not activated or available in the test run.

To enable fallback mode in Sun's reference implementation, provide the `-XX:+FailOverToOldVerifier` option when answering the Other Options question of each interview.

To disable fallback mode in Sun's reference implementation, provide the `-XX:-FailOverToOldVerifier` option when answering the Other Options question of each interview.

5.35.4 Configuration Tab Name-Value Pairs

The Configuration tab name-value pairs in Table 5–74 are provided for use with the JavaTest harness test view and Test Environment dialog box when troubleshooting problems with running tests. You can view the configuration values that were used to run a specific test by choosing the Configuration tab in the Test Manager.

For a description of the test view tabs and Test Environment dialog box, see the *JavaTest Harness User's Guide: Graphical User Interface* included with this release or the JavaTest harness online help.

If you intend to set a configuration value in the command line to run tests, see Chapter 4 for a description of how to obtain the required question-tag names for your configuration.

Table 5–74 lists the name-values pairs displayed in the Configuration tab for the VM tests.

TABLE 5–74 VM Test Configuration Tab Name-Value Pairs

Name	Value Description
`platform.isFallbackSupported`	If your product VM performs the verification by the type inference when the typechecking fails, this value is `true`. If your product VM does not perform the verification by the type inference when the typechecking fails, this value is `false`.

◆ ◆ ◆ CHAPTER 6

6 Debugging Test Problems

Tests fail to execute properly for many reasons. The JavaTest harness provides many tools that you can use to identify the cause of problems when running tests. This chapter provides some approaches for identifying the cause of these problems.

6.1 Test Manager Window

You can quickly view test information in the Test Manager window by clicking folder and test icons in the test tree.

6.1.1 Test Tree

Use the test tree to view the run and test result status of folders and tests in a test suite. The JavaTest harness uses colored icons in the test tree to indicate both the current run status and the test result status.

The goal of a test run is for the root test suite folder to become green, signifying all tests in the test suite that are not filtered out have passed. You can use the filters to specify tests or groups of tests whose results are displayed in the test tree. Using filters when browsing the tree makes it possible to easily monitor folders containing tests that have not passed.

When you click a folder icon in the test tree, the JavaTest harness displays its folder view in the Test Manager information area. The information in the folder view is changed by the view filter.

Refer to the *JavaTest Harness User's Guide: Graphical User Interface* or the JavaTest harness online help for a detailed description.

6.1.2 Folder View

Click a folder icon in the test tree to display its information in the information pane. The folder view displays a Summary tab, five status tabs, and a status display.

During a test run, you can use the folder view to monitor the status of a folder and its tests. You can also use the folder view during troubleshooting to quickly locate and open individual tests that had errors or failed during the test run.

The Summary tab displays the number tests by their test results in a folder. In addition to Summary information, the folder view contains five status tabs that group and list the tests by their results.

The status display at the bottom of the pane displays messages about the selected tab. The messages might indicate that tests in the folder are loading or might provide detailed status information about a selected test.

Refer to the *JavaTest Harness User's Guide: Graphical User Interface* or the JavaTest harness online help for a detailed description.

6.1.3 Test View

To display detailed information about a test, click its icon in the test tree or double-click its test name in the information pane. The information pane displays the test view. The test view contains test information in five tabs and a status message display.

- The Test Description tab displays the name-value pairs contained in the test description.
- The Files tab contains a drop-down list of source files from the test description. You can display the contents of a file by clicking its name in the list.
- The Configuration tab displays a table of JavaTest harness environment values used to run a specific test. The contents are output data and only enabled if the test was run.
- The Test Run Details tab displays the name-value pairs that were recorded when the test was run. The contents are output data and only enabled if the test was run.
- The Test Run Messages tab contains a tree and message panel of output from sections of the test. Click a name to display its contents. The contents are output data and only enabled if the test was run.

Refer to the *JavaTest Harness User's Guide: Graphical User Interface* or the JavaTest harness online help for a detailed description.

6.1.4 Test Manager Properties

You can view the properties of a test manager by choosing View- > Properties from the Test Manager menu bar. The Test Manager Properties dialog box contains Test Suite, Work Directory, Configuration, and Plug-ins information.

Refer to the *JavaTest Harness User's Guide: Graphical User Interface* or the JavaTest harness online help for a detailed description.

6.1.5 Test Suite Errors

If the JavaTest harness detects test suite errors, it displays an advisory dialog box. You can view detailed information about the test suite errors by choosing View -> Test Suite Errors from the Test Manager menu bar.

Refer to the *JavaTest Harness User's Guide: Graphical User Interface* or the JavaTest harness online help for a detailed description.

6.2 JavaTest Harness `ResultBrowser` Servlet

You can use the JavaTest harness `ResultBrowser` servlet included in the `javatest.jar` file to view the contents of a `.jtr` file without starting the JavaTest harness GUI by issuing the following command:

```
com.sun.javatest.servlets.ResultBrowser
```

Refer to the *JavaTest Harness User's Guide: Graphical User Interface* or the JavaTest harness online help for a detailed description.

6.3 Configuration Failures

Configuration failures are difficult to correct. They are easily recognized because many tests fail the same way. When all of your tests begin to fail, you might want to stop the run immediately and start viewing individual test output. However, in the case of full-scale launch problems when tests are not run, individual test output is not available. Also check the following for configuration errors:

- Check your `.jti` file settings in the Configuration Editor to see if you launched the JavaTest harness with Specify Status set to Yes instead of No.
- Although you might not produce an `env.html` report file, the JavaTest harness GUI provides the ability to view evaluated variables used by the tests. Choose Configure- > Show Test Environment from the Test manager menu bar to view the contents of the test environment.
- The Configuration Editor generates a configuration Question Log when you complete a configuration interview. You can use the configuration Question Log to review all of the questions and your answers in the current configuration. Choose Configure -> Show Question Log from the Test Manager menu bar to view the current configuration interview.

6.4 Report Files

You can generate plain text and HTML reports from either the GUI or the command line after a test run. Report files are a good source of information.

Refer to the *JavaTest Harness User's Guide: Graphical User Interface* or the JavaTest harness online help for a detailed description of how to generate and view the test reports.

6.5 Debugging Agent Failures

When debugging agent failures, monitor the JavaTest harness agent during the test run to verify the following:

- The agent is correctly configured and started.
- The test harness is configured correctly for the agent and test system on which the agent started.
- The map files, if required, correctly translate host-specific values into values that the agent can use.

Because active agents initiate the connection with the JavaTest harness while passive agents wait for a request from the JavaTest harness, troubleshooting each type of agent requires a different approach. Refer to the *JavaTest Agent User's Guide* or the JavaTest harness online help for a detailed description of the differences between troubleshooting each type of agent.

6.5.1 Monitoring JavaTest Harness Agents

If you experience problems using JavaTest harness agents to run tests, you can use the JavaTest harness Agent Monitor window to view the agents running tests. You can also use the tabbed panes in the application or applet agent GUI to monitor specific information about the agent and the tests that it runs.

The agent GUI provides a Statistics pane that displays the current status of the tests that the agent is running, a History pane that displays a list of tasks performed by the agent, and a Selected Task pane that displays details about a specific task or test chosen in the history tabbed pane.

Refer to the *JavaTest Agent User's Guide* or the JavaTest harness online help for a detailed description of the JavaTest harness Agent Monitor window and the agent GUI.

6.5.2 `failed.html` Report

The `failed.html` report is an important source of information when debugging test failures. Problems connecting to the JavaTest harness agent are often found in the this file, and might not be reported to a test result file (`.jtr`).

6.5.3 `-trace` Option

You can use the `-trace` option when you start the JavaTest harness agent. The `-trace` option causes the agent application to send detailed information about agent activity to the system output stream.

Refer to the *JavaTest Agent User's Guide* or the JavaTest harness online help for a detailed description of the options that can be used when starting a JavaTest harness agent.

◆ ◆ ◆ CHAPTER 7

7 Workarounds

This section contains information that might be required to run tests on systems with limited resources or that require specific test configurations.

7.1 Running the Agent With Limited Resources

If the system running the JavaTest harness agent has limited resources that prevent mapping a network drive, you might have to load the test classes by using HTTP. You can do this by creating a URL class loader that can load the test classes from the JavaTest harness by way of a web server. An example of a URL Class Loader appears in the AppletViewer source code supplied as part of your Java technology source drop.

7.2 Testing Implementations That Require `System.getProperties()` API

The JavaTest harness security manager denies access to system properties even for the product API under test. This can affect test execution in the single-VM environment (see Chapter 4) or execution of distributed tests (the passive agent for distributed tests also executes a remote test component in single-VM mode).

To run tests for an API that requires access to system properties, you must start the JavaTest harness agent with the `javatest.security.allowPropertiesAccess` property set to `true`. You are not required to set this property for the JavaTest harness agent if you are running tests in multi-VM mode with an agent (see Chapter 4).

For example, if you run the JMX Remote API tests (tests for the `javax.management.remote` API) on Sun's reference runtime in single-VM mode the following VM option must be specified for both VMs running the JavaTest harness agent and the passive agent for distributed tests:

```
-Djavatest.security.allowPropertiesAccess=true
```

If you run the same tests in multi-VM mode set this option only for the passive agent.

◆ ◆ ◆ APPENDIX A

JCK Test Specification

This appendix describes the structure of JCK tests and how they are executed. The appendix is divided into the following two sections:

- "A.1 Test Description" on page 227 — Contains the information that describes a JCK test.
- "A.2 Test Selection and Execution" on page 241 — Describes how tests are selected for a test run and the steps involved in executing compiler and runtime tests using the information in the test description.

A.1 Test Description

Every test in the JCK test suite has a corresponding test description. The test description is an HTML file or files that contain a test description table and test comments. Test descriptions can be organized in the following ways:

- Test description table and comments for a single test contained in a single file
- Test description tables and comments for a number of similar tests grouped together in a single file
- Test description tables and comments in separate files linked together using hypertext links

Test descriptions can be viewed using a web browser.

A.1.1 Test Comments

Because it is formatted using HTML, the test description is easily read by computer programs, test developers, and users. All test descriptions in the JCK test suite contain comments that describe what the test does and how it works.

A.1.2 Test Description Table

The test description file contains the test description table. The test description table is an HTML table. Each row of the table contains a test description field. The values specified in the test description fields are used by the JavaTest harness when the tests are run.

See Table A–1 for an example of a test description table.

TABLE A–1 Sample Test Description Table

Item	Value
`title`	`Checkbox - Checkbox Tests`
`name`	`Checkbox`
`source`	`CheckboxTests.java`
`executeClass`	`javasoft.sqe.tests.api.java.awt.Checkbox.CheckboxTests`
`keywords`	`runtime positive`
`executeArgs`	`-TestCaseID ALL`

Table A–2 describes the test description fields used by JCK 6b:

TABLE A–2 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.
`name`	A string that identifies the test.

TABLE A–2 Test Description Fields *(Continued)*

Field	Description
`source`	For compiler tests, the source field contains the names of the files that are compiled during the test run. 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 JCK. Source files are included for reference only. Source files are often `.java` files, but can also be `.jasm` or `.jcod` files. ■ `.jasm` is a low level bytecode assembler designed to assemble class files containing sets of bytecodes that are unusual or invalid for use in runtime tests ■ `.jcod` is a class level assembler designed to build classes with unusual or invalid structure for use in runtime tests These tools are used to generate class files that cannot be reliably generated by a Java compiler. For most XML parser tests in the JCK runtime, the source field contains the names of the files that are processed during the test run. These files are XML Schema sources and XML documents usually having file name extensions of `.xsd` and `.xml` respectively. Such tests share a single precompiled class, `TestRun`, that invokes the XML parser under test through the Java technology API and passes the source file names to the parser for processing. The test model is similar to compiler testing because the sources used in the tests contain valid and invalid use of various constructs of corresponding languages.
`keywords`	String tokens that can be associated with a given test. They describe attributes or characteristics of the test (for example, how the test should be executed, and whether it is a positive or negative test). Keywords are often used to select or deselect tests from a test run. Keywords are described in detail in "A.1.3 Keywords" on page 231.
`executeClass`	The name of the main tests's class that the JavaTest harness loads and runs. This class might 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 a few JNI implementation tests that launch a VM from native C code.

TABLE A–2 Test Description Fields *(Continued)*

Field	Description
`executeArgs`	An array of strings that are passed to the test classes being executed. The arguments might be fixed but often involve symbolic values that are substituted from the test environment (variables defined elsewhere in the test environment). The result of substituting values can be seen in the resulting `.jtr` files. These arguments form the basis for the set of arguments that are passed into the tests defined in `executeClass` and `executeNative` fields. The default value of any variable not defined in the test environment is an empty string.
`rmicClasses`	For Java RMI tests, this nominates the set of classes that are passed to the Java RMI compiler.
`timeout`	A value specified in seconds used to override the default ten minute timeout used with all JCK tests.
`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 passes the values through to the test. If any of these values are not defined, the JavaTest harness does not run the test. Context-sensitive properties are described in detail in "A.1.4 Context-Sensitive Properties" on page 233.

`selectIf`	Specifies a condition that must be satisfied for the test to be executed. This field is constructed using environment values, Java programming language literals, and the full set of Boolean operators:
	
	`(+, -, *, /, <, >, <=, >=, &, |, !, !=, ==).`
	
	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 values are not defined in the test environment, the JavaTest harness considers the test to be in error.

`remote`	This entry contains information required to run distributed network tests.

TABLE A–2 Test Description Fields *(Continued)*

Field	Description
`jplisAgent`	Specifies one Java PLIS agent. Following is the field value format: *agent-class-name options* ■ *agent-clas-name* is the fully-qualified class name of the JPLIS agent. ■ *options* is a nonblank-space separated string that represents options to be passed to the agent specified by `agent-class-name`. One test description table might be several `jplisAgent` fields. The `jplisAgent` field is processed only if the executed test has the `jplis` keyword in the list of keywords in the test description table.
`apClass`	The name of the annotation processor tests' class that is passed to the tested Java compiler.
`apSource`	Contains the names of files previously compiled to create the annotation processor's class files. The precompiled class files are included with the JCK.
`apArgs`	The annotation processor specific option value that is passed to the tested Java compiler. The value of this field is stored in the `jckApArgs` option name. The following is an example of passing the option with the standard JDK on Solaris: `-AjckApArgs=$apArgs`.

A.1.3 Keywords

Keywords are tokens associated with specific tests. Keywords have two functions:

- Convey information to the JavaTest harness about how to execute the tests.
- Serve as a basis for including and excluding tests during test runs. You can specify keyword expressions in the JavaTest harness Configuration Editor keywords field to filter tests during test runs (see Chapter 4).

Keywords are specified in the keywords field of the test description. The set of valid keywords for JCK 6b is described in Table A–3 and Table A–4.

Table A–3 describes JCK keywords used in a wide variety of tests.

TABLE A–3 General Keywords

Field	Description
`compiler`	Identifies tests used with compiler products.
`interactive`	Identifies tests that require human interaction.
`negative`	The component under test must terminate with (and detect) an error. An operation performed by a negative test on the component under test must not succeed.
`only_once`	Identifies tests that can only be run one time in the same VM. While tests are usually run once, it is possible that tests could be run more often. Tests with this keyword fail to run correctly if run more than once in the same VM.
`positive`	The component under test must terminate normally. An operation performed by the test on the component under test must succeed.
`robot`	If your system supports the `awt.robot` API, the tests can be run automatically. Otherwise, they run interactively.
`runtime`	Identifies tests used with runtime products.
`serial`	Used only with compiler tests. Identifies a test whose source files must be compiled one at a time by separate compiler invocations. If `serial` is not specified, the source files are compiled by group (`.java`).

Table A–4 describes keywords used only in specific tests.

TABLE A–4 Test Specific Keywords

Field	Description
`distributed`	Identifies distributed tests that validate functionality while the test platform is connected to a remote host.
`idl_inherit`	Identifies tests that require the IDL-to-Java programming language translator to output Java technology interfaces that use the inheritance object model.
`idl_tie`	Identifies tests that require the IDL-to-Java programming language translator to output Java technology interfaces that use the TIE object model.
`java_to_schema`	Identifies JAXB tests that validate Java-to-Schema mapping.

TABLE A–4 Test Specific Keywords *(Continued)*

Field	Description
`jaxb`	Identifies JAXB tests that validate API and Java-to-Schema mapping.
`jaxws`	Identifies tests that validate JAX-WS functionality.
`jniinvocationapi`	Identifies tests that use the JNI implementation.
`jplis`	Identifies JPLIS tests that require separate execution in single-VM mode and special options in the tested Java technology command line.
`jplislivephase`	Identifies JPLIS Live Phase tests that require separate execution in single-VM mode and that require support of an optional mechanism to start instrumentation agents some time after the VM has started without specifying agents through Java platform command line options.
`jvmti`	Identifies JVM TI tests that require separate execution in single-VM mode and special options in the tested Java technology command line.
`jvmtilivephase`	Identifies JVM TI Live Phase tests that require separate execution in single-VM mode and that require support of an optional mechanism to start instrumentation agents some time after the VM has started without specifying agents through Java platform command line options.
`jdwp`	Identifies JDWP tests that require separate execution in single-VM mode and special options in the tested Java technology command line.
`needsharedclassloader`	Identifies tests that must be loaded by the same class loader when they are run in single-VM mode. Tests with this keyword must use the same native library due to the Java SE limitation that does not allow the same native library to be loaded by different class loaders.
`rmi_v11`	Identifies RMI compiler tests that require version 1.1 stubs or skeletons to be produced by the RMI compiler.
`rmi_iiop`	Identifies RMI tests that use the IIOP transport layer.

A.1.4 Context-Sensitive Properties

A number of tests require information about the particular Java platform technology under test (context) to determine expected results. This information is identified by entry names in the `context` field of the test description table. The JavaTest harness checks to be sure that all values

specified in the `context` field are defined in the test environment before it attempts to execute the test. If any of the values are not set, the test is not executed and the test is considered to be in error.

The valid context-sensitive properties for JCK 6b are described in the following sections.

A.1.4.1 Network Resources

Table A–5 lists network resources, which are properties that the network tests use, such as the IP addresses of systems used during the tests.

TABLE A–5 Network Resources

Field	Data Format	Description
`network.fileURL`	URL	A file URL for the network tests. For example: `file:///C:\\AUTOEXEC.BAT`
`network.ftpURL`	URL	An FTP URL for the network tests. For example: `ftp://javator/etc/hosts`
`network.httpURL`	URL	An HTTP URL for the network tests to use. For example: `http://www.sun.com`
`network.jckHTTPProxyHost`	Proxy name	The name of the HTTP proxy server to be used by JCK tests if a direct connection is not used. For example: `fproxy.somewhere.com`
`network.jckHTTPProxyPort`	Port number	The port number of the HTTP proxy server to be used by JCK tests if a direct connection is not used. For example: `8080`
`network.localHost`	Hostname and IP address	The name and IP address of the local host. For example: `ruffles/129.144.75.50`

TABLE A–5 Network Resources *(Continued)*

Field	Data Format	Description
`network.tcpPortRange`	Range of ports	On some systems, the set of ports allocated by the OS is different from the set of ports that can be specifically requested by client code. In this case, `network.tcpPortRange` must be set to indicate to the test the range of ports that are available. The lower limit is 1024 and the upper limit is 65535.`network.tcpPortRange` test configuration value For example: `2048-5096`
`network.testHost1`	Hostname and IP address	A host used for network tests. For example: `satchmo/129.144.250.133`
`network.testHost2`	Hostname and IP address	Another test host (different from testHost1). For example: `doppio/129.144.173.35`
`network.udpPortRange`	Range of ports	On some systems, the set of ports that are allocated by the OS is different from the set of ports that can be specifically requested by client code. In this case, `network.udpPortRange` must be set to indicate to the test the range of ports that are available. The lower limit is 1024 and the upper limit is 65535. For example: `2048-5096`

A.1.4.2 Remote Resources

Many network tests communicate with an additional system (distributed test server) during the test run. These tests are referred to as distributed tests. Table A–6 identifies and describes the configuration values that must be set for the distributed tests to work.

TABLE A–6 Remote Resources

Name	Data Format	Description

`remote.networkAgent`	`-host` *hostname*
	`-port` *port number* (optional)
	`-classpath` *classpath* (optional)
		The distributed server to use with distributed tests.
		For example:
		`-host doppio`
		`-port 345`
		`-classpath D:\classess`

A.1.4.3 Hardware Characteristics

Table A–7 identifies and describes the features that differ based on hardware characteristics.

TABLE A–7 Hardware Characteristics

Name	Format	Description
`hardware.xFP_ExponentRanges`	Integer	Runtime argument for floating-point tests. For more information, see Chapter 5. For example: `-16382:16383:-16382:16383`[1]

[1] *Note:* These values are only valid for Java technology runtime implementations that use only the standard formats for floating-point operations.

A.1.4.4 Platform Characteristics

Table A–8 identifies and describes the features that might or might not be present on a particular hardware and software platform.

TABLE A–8 Platform Characteristics

Name	Format	Description
`platform.canPlayMidi`	Boolean	Specifies if the system can play MIDI encoded sound. For example: `false`
`platform.canPlaySound`	Boolean	Specifies if the system supports the ability to play sampled audio. For example: `true`

TABLE A–8 Platform Characteristics *(Continued)*

Name	Format	Description
`platform.canRecordSound`	Boolean	Specifies if the system supports the ability to record sound. For example: `true`
`platform.expectOutOfMemory`	Boolean	Indicates if the tested system can throw an `OutOfMemoryError`. For example: `true`
`platform.hasPrinter`	Boolean	Indicates if a printer is connected to the tested system. If a printer is connected to the system and is online, the value must be set to `true`. For example: `true`
`platform.isFallbackSupported`	Boolean	Indicates if the system performs the verification by the type inference when the typechecking fails. For example: `true`
`platform.isHeadless`	Boolean	Indicates if the tested system is headless (does not have a display device, a keyboard, or mouse). For example: `false`
`platform.JavaSecurityAuthPolicy`	String	Specifies the mechanism your product uses to specify the Java platform Security Authorization Policy used to run JAAS tests. For example: `java.security.auth.policy`

TABLE A–8 Platform Characteristics *(Continued)*

Name	Format	Description
`platform.JavaSecurityAuthLoginConfig`	String	Specifies the mechanism your product uses to specify the Java platform Security Login Configuration used to run JAAS tests. For example: `java.security.auth.login.config`
`platform.JavaSecurityKrb5KDC`	String	VM System Property to set the Java platform Security Kerberos Key Distribution Center. For example: `java.security.krb5.kdc`
`platform.JavaSecurityKrb5Realm`	String	VM System Property to set the Java platform Security Kerberos Key Distribution Center realm. For example: `java.security.krb5.realm`
`platform.KDCHostName`	String	Host name of the Kerberos Key Distribution Center. For example: `plop.sfbay.sun.com`
`platform.jdwpSupported`	Boolean	Indicates if the JDWP implementation is provided on your system. For example: `true`
`platform.jvmtiSupported`	Boolean	Indicates if the system supports JVM TI. For example: `true`
`platform.MultiVM`	Boolean	Specifies if your system can load and run separate instances of a tested VM. For example: `false`

TABLE A–8 Platform Characteristics *(Continued)*

Name	Format	Description
`platform.KDCRealm`	String	Kerberos realm name containing the following two accounts that are operating. For example: `PLOP`
`platform.KerberosClientPassword`	String	Password for the second principal. For example: `xxxxxxx`
`platform.KerberosClientUsername`	String	Second Kerberos principal name. For example: `client`
`platform.KerberosServerUsername`	String	First Kerberos principal name. For example: `server`
`platform.KerberosServerPassword`	String	Password for the first principal. For example: `xxxxxxx`
`platform.Krb5LoginModuleConfigured`	Boolean	Indicator whether module `com.sun.security.auth.module.Krb5LoginModule` is configured in the JRE software. For example: `true`
`platform.maxMemory`	Positive double or zero with optional postfix m or k	Amount of memory a test might allocate (in megabytes, kilobytes, or bytes). For example: `205m`
`platform.nativeCodeSupported`	Boolean	Indicates if the system supports direct access to native code (including but not limited to JNI implementation). For example: `true`

TABLE A–8 Platform Characteristics *(Continued)*

Name	Format	Description
`platform.robotAvailable`	Boolean	Indicates if the system supports low-level input control (provided by `java.awt.robot` classes) For example: `true`
`platform.rowSetFactory`	String	The fully qualified name of the factory class used for creating the rowsets.

`latform.soundURL`	URL	The URL path to a sound file available to the system. The URL is used by various sound tests for playback and conversion testing. The URL should be a sound file (`wav`, `au`, or `snd`) that your system is capable of playing.
		Example:
		`file:`
		`$testSuiteRootDir$/api$/javax_sound $/sound.wav`

`platform.staticSigTestClasspath`	Classpath	List of directories and ZIP or JAR files where the classes tested by the Static Signature Test are located, separated with a system-dependent path separator. Example: `/usr/local/java/solsparc/jre /lib/rt.jar`
`platform.supportStaticSigTest`	Boolean	Boolean value that indicates whether the static signature test is executed on the system being tested. The static signature test must be executed if the class files for the JRE software are delivered in a directory hierarchy, or in a ZIP or JAR file. Example: `true`

A.2 Test Selection and Execution

This section describes how JCK tests are selected for a test run and how they are then executed.

A.2.1 Test Results

Table A–9 describes the three states used to report test execution results.

TABLE A–9 Test Execution Result States

State	Description
Pass	A test passes when the functionality being tested behaves as expected. All tests are expected to have passing results.
Fail	A test has failing results 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. A single configuration problem can cause many tests to fail. For example, if the path to the runtime is configured incorrectly, the harness cannot run tests and identifies them as having errors.

A.2.2 Tests Selected for a Test Run

Table A–10 describes the factors used by the JavaTest harness to select tests immediately prior to starting a test run.

TABLE A–10 Test Selection Factors

Selection Factor	Description
Tests (JavaTest harness GUI)	The JavaTest harness finds tests listed in the Tests to be Run field of the Configuration Editor. You can specify sub-branches of the tree as a way of limiting the tests 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. See Chapter 4 for more information.
Exclude list (JavaTest harness GUI)	Tests listed in the appropriate exclude list are deselected prior to the start of a test run. For details about exclude lists and their role in the certification process, see Chapter 2. For a description about how to specify an exclude list using the Configuration Editor, see Chapter 4.

TABLE A–10 Test Selection Factors *(Continued)*

Selection Factor	Description
Keywords (JavaTest harness GUI)	A test can be selected based on keywords specified in the keywords field in the test description. JCK keywords are described in detail in "A.1.3 Keywords" on page 231. For a description about how to use keywords with the JavaTest harness Configuration Editor, see Chapter 4.
Prior Status	Use the checkboxes to select tests in a test run based on their outcome on a prior test run. Prior status is evaluated on a test-by-test basis using information stored in result files (`.jtr`) written in the work directory.

`selectIf` field (Test Description table)	The `selectIf` field in the test description table contains an expression composed of elements from the test environment, Java programming language literals, and the standard operators:
	`+, -, *, /, <, >, <=, >=, &, |, !, !=, ==`
	For 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.

A.2.3 Compiler Test Execution

The section describes how JCK compiler tests are executed. Compiler tests contain the compiler keyword in the test description. They test either the Java compileror the RMI compiler.

A.2.3.1 Java Compiler Tests

All Java compiler test source files are `.java` files.

If the `apClass` field is specified in the test description, the annotation processor class name in this field is passed to the Java compiler and to those test source files not specified in the `apSource` test description field.

If the test description keywords field contains the `serial` keyword, the source files are compiled one at a time in the order they are listed in the test description source field. If the `serial` keyword is not specified, they are compiled as a group.

If the `negative` keyword is specified in the test description, the test acquires the following result status:

- *Fails* if the compilation succeeds

- *Passes* if the compilation fails

If the `positive` keyword is specified in the test description, the test acquires the following result status:

- *Fails* if the compilation does not succeed
- If the compilation succeeds, the resulting classes are executed on a reference runtime, and the test acquires the following result status:
  - *Fails* if execution does not succeed
  - *Passes* if execution succeeds

Note – Compilation is carried out using the `command.testCompile.java` entry from the test environment. The variable `$testSources` is set to the source files listed in the test description source field.

Note – Classes are executed using the command `command.refExecute`. The `$testExecuteClass` variable is set to the class to execute, the `$testExecuteArgs`variable is set to the value of the arguments to be passed to the class to execute. See "A.2.4.1 `testExecuteArgs` Variable" on page 251 for more information.

Figure A–1 illustrates the execution model.

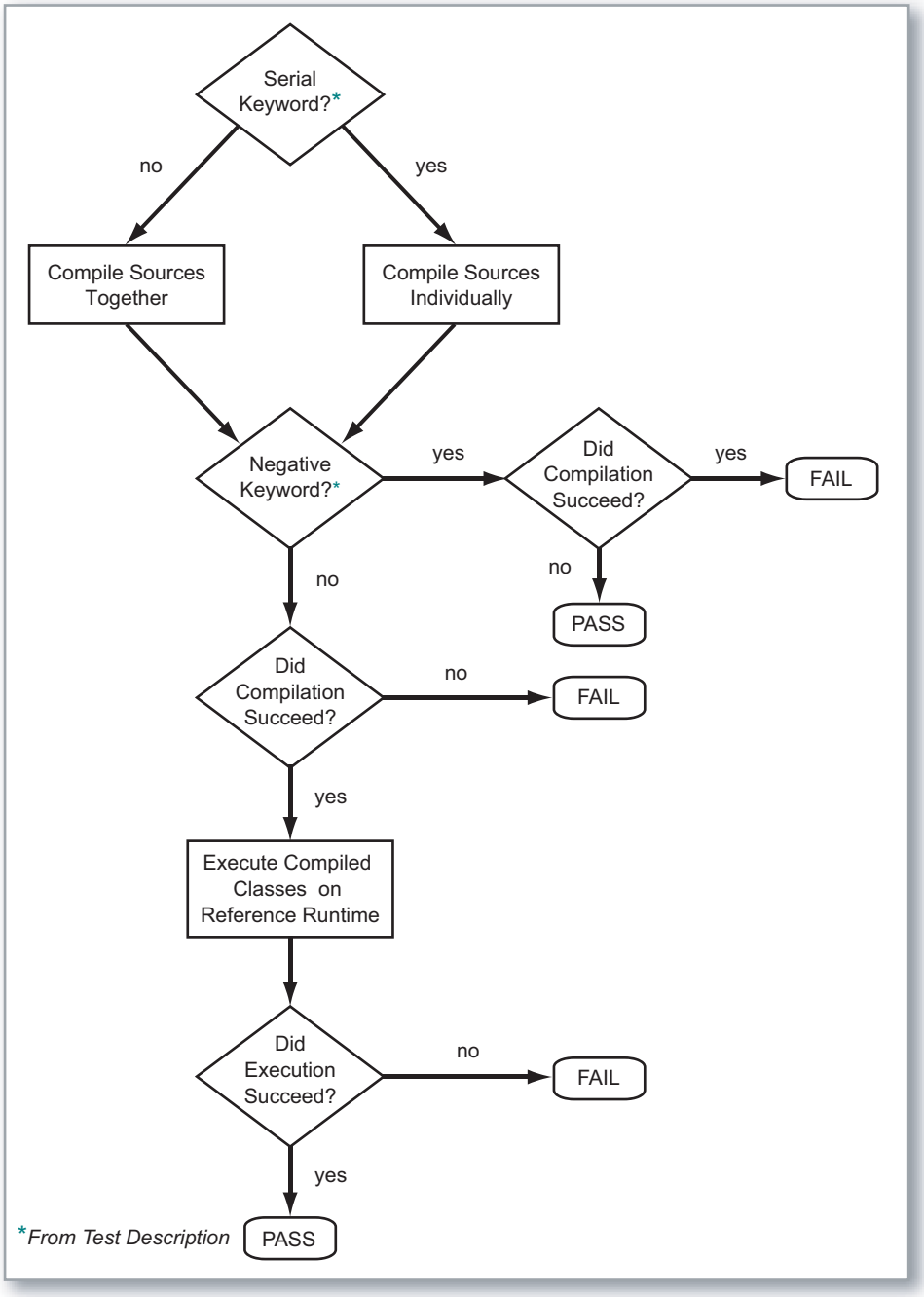


FIGURE A–1 Java Class File Compilation Test Flow Diagram

A.2.3.2 RMI Compiler Tests

Two steps are performed for RMI compiler (RMIC) tests.

1. Java programming language sources are compiled into Java class files.

   If the Java programming language compilation does not succeed, the RMIC step is not run and the JavaTest harness considers the test to be in error.

   Note – For notes about compiling Java class files, see "A.2.3.1 Java Compiler Tests" on page 242.

2. Some or all of the resulting class files are then compiled by the RMIC.

   The test description field of the `rmicClasses` contains the list of classes to be compiled by the RMIC.

   If the `negative` keyword is specified in the test description, the test acquires the following result status:

   - *Fails* if the RMIC compilation succeeds
   - *Passes* if the RMIC compilation does not succeed

   If the `positive` keyword is specified in the test description and the compilation does not succeed, the test *fails*.

   If the `positive` keyword is specified in the test description and the compilation succeeds, the resulting class files are executed on a Reference Runtime. The test acquires the following result status:

   - *Fails* if execution does not succeed
   - *Passes* if execution succeeds

   The RMIC compilation is carried out using one of two commands.

   If the `rmi-iiop` keyword is specified, use the following command:

   ```
   command.testRMIC.iiop
   ```

   If the `rmi-iiop` keyword is not specified, use the following command:

   ```
   command.testRMIC
   ```

   The list of class files to be compiled is derived from the list of class files specified in the `rmicClasses` test description field and is contained in the `$testRmicClasses` variable.

   Figure A–2 illustrates the execution model.

Note – Test class files are executed using the command `command.refExecute`. The `$testExecuteClass` variable is set to the class file to execute, the `$testExecuteArgs` variable is set to the value of the arguments to be passed to the class file to execute. See "A.2.4.1 `testExecuteArgs` Variable" on page 251 for more information.

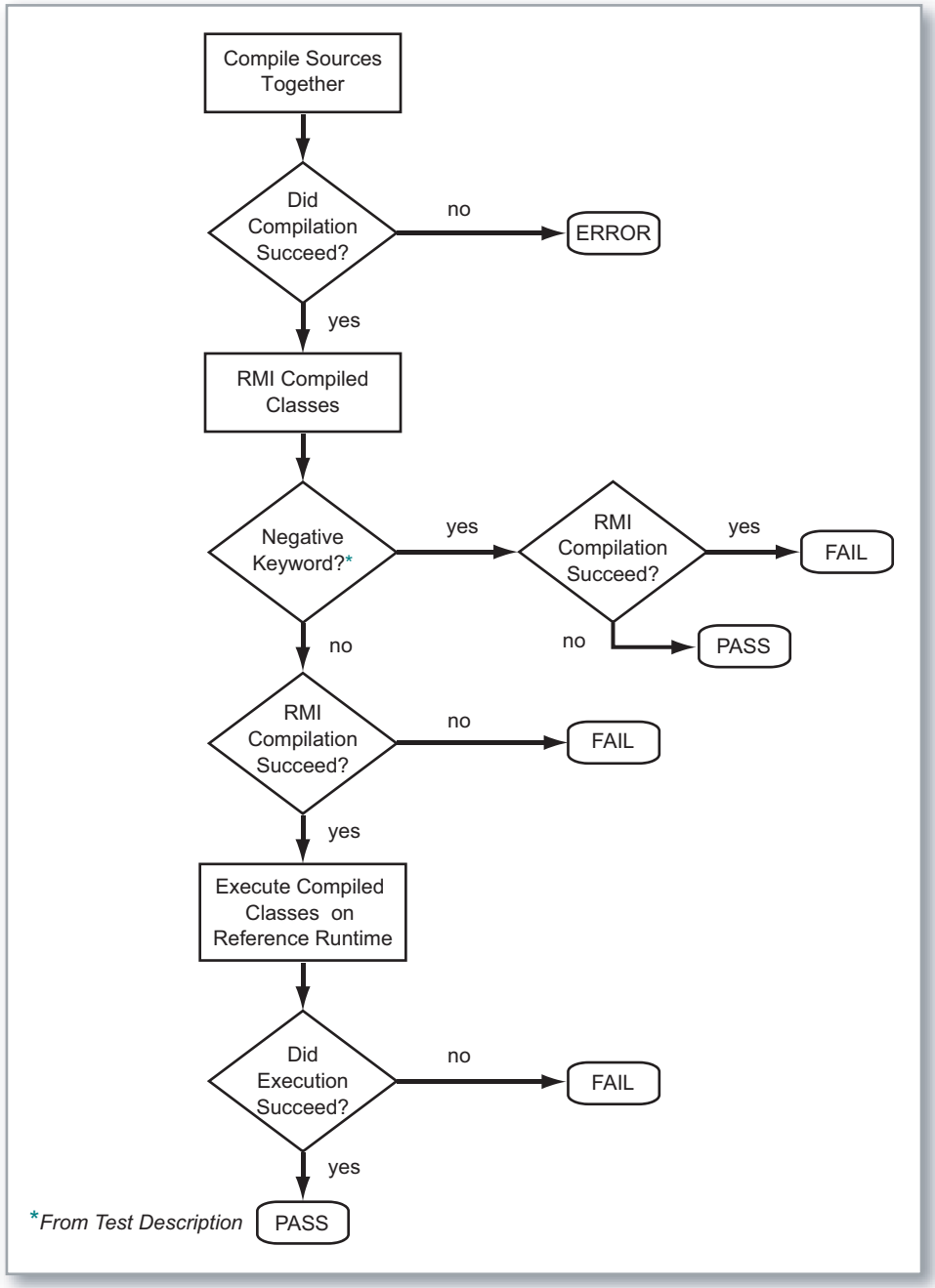


FIGURE A–2 RMI Compilation Test Flow Diagram

A.2.4 Runtime Tests

Runtime tests are all tests that contain the `runtime` keyword in the test description.

Runtime tests might be written in the Java programming language or in one of the two byte code assemblers (`.jasm` and `.jcod`) used to construct class files with specific characteristics:

- `.jasm` is a low-level bytecode assembler designed to assemble sets of bytecodes that are unusual or invalid for use in runtime tests.
- `.jcod` is a class-level assembler designed to build invalid classes for use in runtime tests.

Runtime tests are packaged as compiled class files although the sources are also included for reference purposes.

▼ Executing Runtime Tests

1 If the test description table contains remote fields, the test is a distributed test, the following applies:

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. The value of `$testMsgSwitch` is set to the name of the machine running the JavaTest harness and a port on that machine to use for the communication:

```
$testMsgSwitch=host:port
```

b. Each of the remote entries is activated.

Note – For more information about the remote field, see Table A–2.

---

2 The main class file 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 is an unintended result, as opposed to a failure that could be an intended result (see Step 4).

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 file.

The final status is the "first worst" status: failed is worse than pass, error is worse than failed).

4 Positive and negative checks.

If the test description contains the `negative` keyword the test acquires the following result status:

- *Passes* if execution fails (Step 2). Few negative runtime tests are used. A few tests 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 for local tests, Step 3 for distributed tests).

Note – Test class files are executed using the command `command.testExecute`. The `$testExecuteClass` variable is set to the class file to execute, the `$testExecuteArgs` variable is set to the value of the arguments to be passed to the class file to execute. See "A.2.4.1 `testExecuteArgs` Variable" on page 251 for more information.

Figure A–3 illustrates the execution model.

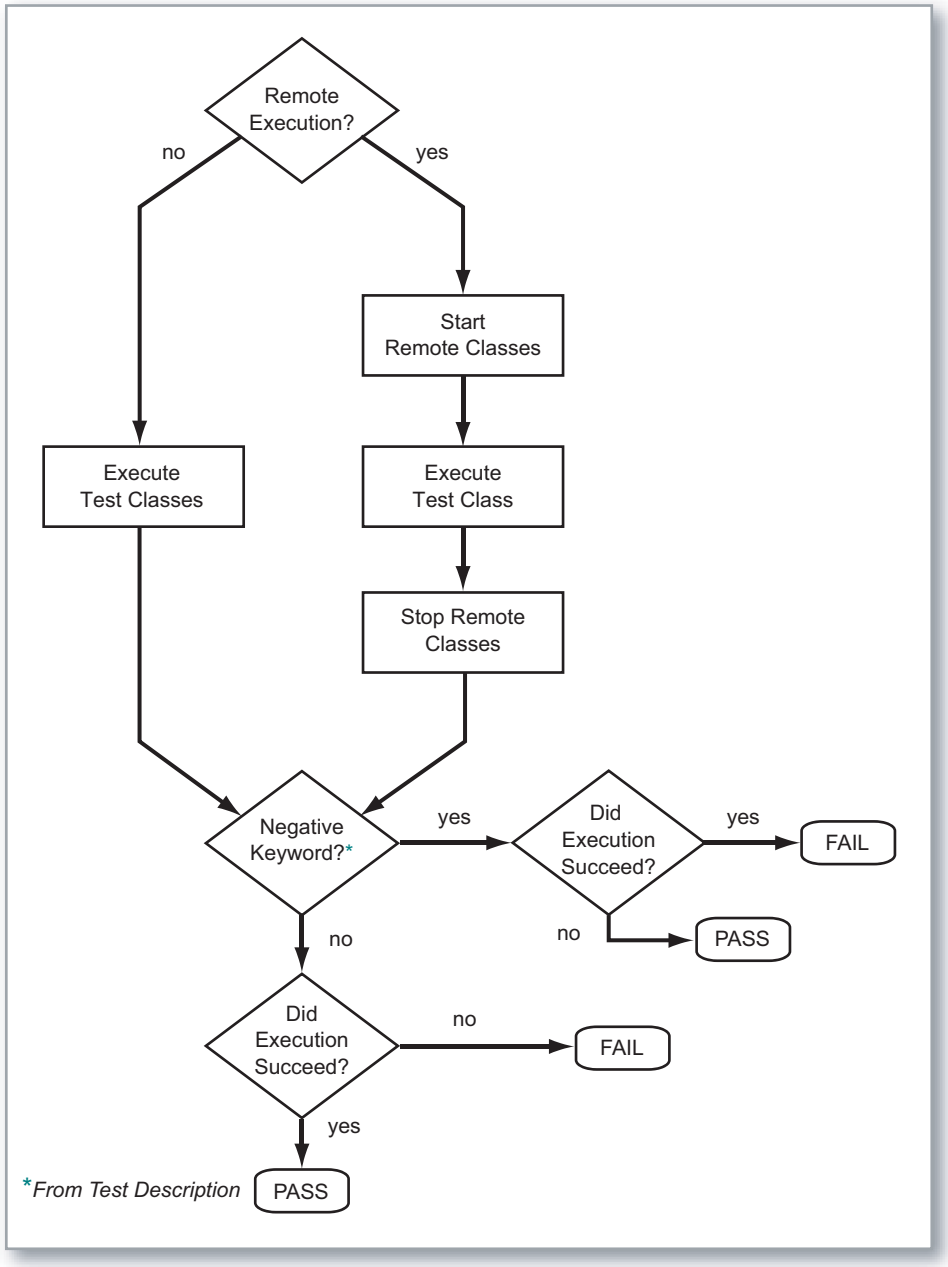


FIGURE A–3 Runtime Test Flow Diagram

A.2.4.1 `testExecuteArgs` Variable

Whenever the JavaTest harness executes a test class file it obtains arguments required by the class file (if any exist) from the `$testExecuteArgs` variable in the test environment. The `$testExecuteArgs` variable is constructed from a variety of sources and assembled in the following order:

1. Some runtime tests include a number of test cases. Some of these tests might be excluded. If the exclude list identifies cases to be excluded, these cases are passed into the test using the `-exclude` option.

   For example:

   ```
   -exclude testcase1,testcase2,...
   ```

2. For each entry *EEE* in the context field of the test description table, the JavaTest harness looks up *EEE* in the environment. If the name is not found, the test is automatically in *error*, otherwise `$testExecuteArgs` has an entry equivalent to the following:

   ```
   -EEE $EEE
   ```

3. Any values supplied to the `executeArgs` field of the test description table are added to `$testExecuteArgs`. These values have any `$...` entries resolved in the environment, but no errors occur if any of these values are not defined.

◆ ◆ ◆ APPENDIX B

Detailed Examples

The following examples are provided to assist the user in setting up tests to run in specific testing environments. Before using these examples, you must modify them for your specific testing environment.

B.1 Running JCK Tests Using Multiple VMs

The following procedures demonstrate how to set up and run JCK 6b tests for the following test environments and configurations:

- "Run JCK Runtime Tests Locally on a Windows System" on page 253
- "Run JCK Runtime Tests Locally on a Solaris System" on page 255
- "Run JCK Runtime Tests Remotely on a Windows System" on page 257
- "Run JCK Runtime Tests Remotely on a Solaris System" on page 259

▼ Run JCK Runtime Tests Locally on a Windows System

1 **Complete all required special test setup steps.**

See "4.5 Special Setup Instructions" on page 104.

2 **Start the JavaTest harness.**

When starting the harness, you must allocate additional memory for opening the JCK test suite. Use the following command to start the harness from the directory where the `javatest.jar` file is located:

```
java -Xmx512m -XX:PermSize=128m -XX:MaxPermSize=128m -jar javatest.jar
```

Note – When allocating memory to the VM, do not exceed the actual amount of memory available on your system.

If you start the harness from a directory that does not contain the `javatest.jar` file, you must include its path in the command. For example:

```
java -Xmx512m -XX:PermSize=128m -XX:MaxPermSize=128m -jar c:\jck\sampleJCK-runtime-6b\lib\javatest.jar
```

In the example, the harness is started from the root directory of the `c:` drive. The `javatest.jar` file is located on the `c:` drive in the *jck*`\sampleJCK-runtime-6b\lib` directory. Change these values for your specific installation.

Note – The default RI PermGen values can be not large enough to perform a complete JCK run. 64-bit platforms require 256m permanent generation memory, whereas 32-bit platforms require 128m permanent generation memory. Include the appropriate options in the command to run tests for your test platform.

---

3 Use the Quick Start wizard to specify the test suite, work directory, and configuration loaded in the Test Manager window.

Note – If the harness opens the GUI without displaying the Quick Start wizard, launch the Quick Start wizard by choosing File -> Open Quick Start Wizard from the menu bar.

---

a. When the Quick Start wizard opens, click the Start a new test run radio button and the Next button.

b. Click the Create a new configuration radio button and the Next button.

If your group does not use configuration templates, you must use an empty JCK interview to create a configuration for the test run.

If your group uses configuration templates, you can load a template in the Quick Start wizard and use it to create the configuration used for the test run.

c. In the Test Suite pane, choose the test suite that contains the tests that you intend to run.

Use the `sampleJCK-runtime-6b` test suite for this example.

d. In the Work Directory pane, create a work directory for the test suite.

The work directory contains all of the information collected by the JavaTest harness during the test run of this test suite and configuration combination.

e. Select the Start the Configuration Editor checkbox and the Start test run checkbox.

The Quick Start wizard automatically opens the Configuration Editor for you when it closes. When you complete the configuration and close the Configuration Editor, the harness automatically begins the test run.

f. Click the Finish button.

4 Create a configuration for the test run.

Each interview contains the following three major sections:

- The first section of each interview contains a series of sub-interviews that collect the information required to configure a test environment.
- The second section of the interview collects information about the tests from the JCK 6b test suite that is to be run.
- The last section of the interview collects information about how the tests are run (such as keywords and the exclude list) and contains JCK 6b specific information and values.

a. In the first section of the interview, provide the answers from the More Information table below.

b. Answer the remaining interview questions with settings required for your test system.

5 Click the Done button.

Because you selected the Start test run checkbox in the Quick Start wizard, the test run begins automatically when the Configuration Editor closes.

More Information Interview Responses for Runtime Test Run Locally on a Windows System Using Multiple VMs

Interview Question	Answer
Test Execution Mode — Select the test execution mode	`MultiJVM`
Testing Locally - Are you running these tests on the same system used for this interview?	`Yes`

▼ Run JCK Runtime Tests Locally on a Solaris System

1 Complete all required special test setup steps.

See "4.5 Special Setup Instructions" on page 104.

2 Start the JavaTest harness.

When starting the harness, you must allocate additional memory for opening the JCK test suite. Use the following command to start the harness from the directory where the `javatest.jar` file is located:

```
java -Xmx512m -XX:PermSize=256m -XX:MaxPermSize=256m -jar javatest.jar
```

Note – When allocating memory to the VM, do not exceed the actual amount of memory available on your system.

If you start the harness from a directory that does not contain the `javatest.jar` file, you must include its path in the command. For example:

```
java -Xmx512m -XX:PermSize=256m -XX:MaxPermSize=256m -jar /jck6b/sampleJCK-runtime-6b/lib/javatest.jar
```

In the example, the harness is started from the root directory. The `javatest.jar` file is located in the `jck6b/sampleJCK-runtime-6b/lib` directory. Change these values for your specific installation.

3 Use the Quick Start wizard to specify the test suite, work directory, and configuration loaded in the Test Manager window.

Note – If the harness opens the GUI without displaying the Quick Start wizard, launch the Quick Start wizard by choosing File -> Open Quick Start Wizard from the menu bar.

---

a. When the Quick Start wizard opens, click the Start a new test run radio button and the Next button.

b. Click the Create a new configuration radio button and the Next button.

If your group does not use configuration templates, you must use an empty JCK interview to create a configuration for the test run.

If your group uses configuration templates, you can load a template in the Quick Start wizard and use it to create the configuration used for the test run.

c. In the Test Suite pane, choose the test suite that contains the tests that you intend to run.

Use the `sampleJCK-runtime-6b` test suite for this example.

d. In the Work Directory pane, create a work directory for the test suite.

The work directory contains all of the information collected by the JavaTest harness during the test run of this test suite and configuration combination.

e. Select the Start the Configuration Editor checkbox and the Start test run checkbox.

The Quick Start wizard automatically opens the Configuration Editor for you when it closes. When you complete the configuration and close the Configuration Editor, the harness automatically begins the test run.

f. Click the Finish button.

4 Create a configuration for the test run.

Each configuration interview contains the following three major sections:

- The first section of each interview contains a series of sub-interviews that collect the information required to configure a test environment.
- The second section of the interview collects information about the tests from the JCK 6b test suite that is to be run.
- The last section of the interview collects information about how the tests are run (such as keywords and the exclude list) and contains JCK 6b specific information and values.

  a. In the first section of the interview, provide the answers from the More Information table below.

  b. Answer the remaining interview questions with settings required for your test system.

5 Click the Done button.

Because you selected the Start test run checkbox in the Quick Start wizard, the test run begins automatically when the Configuration Editor closes.

More Information Interview Responses for Runtime Test Run Locally on a Solaris System Using Multiple VMs

Interview Question	Answer
Test Execution Mode — Select the test execution mode	`MultiJVM`
Testing Locally - Are you running these tests on the same system used for this interview?	`Yes`

▼ Run JCK Runtime Tests Remotely on a Windows System

1 Complete all required special test setup steps.

See "4.5 Special Setup Instructions" on page 104.

2 Start the JavaTest harness.

When starting the harness, you must allocate additional memory for opening the JCK test suite. Use the following command to start the harness from the directory where the `javatest.jar` file is located:

```
c:\>java -Xmx512m -XX:PermSize=128m -XX:MaxPermSize=128m -jar javatest.jar
```

Note – When allocating memory to the VM, do not exceed the actual amount of memory available on your system.

If you start the harness from a directory that does not contain the `javatest.jar` file, you must include its path in the command. For example:

```
c:\>java -Xmx512m -XX:PermSize=128m -XX:MaxPermSize=128m -jar c:\jck\sampleJCK-runtime-6b\lib\javatest.jar
```

In the example, the harness is started from the root directory of the `c:` drive. The `javatest.jar` file is located on the `c:` drive in the *jck*`\sampleJCK-runtime-6b\lib` directory. Change these values for your specific installation.

3 Use the Quick Start wizard to specify the test suite, work directory, and configuration loaded in the Test Manager window.

Note – If the harness opens the GUI without displaying the Quick Start wizard, launch the Quick Start wizard by choosing File -> Open Quick Start Wizard from the menu bar.

---

a. When the Quick Start wizard opens, click the Start a new test run radio button and the Next button.

b. Click the Create a new configuration radio button and the Next button.

If your group does not use configuration templates, you must use an empty JCK interview to create a configuration for the test run.

If your group uses configuration templates, you can load a template in the Quick Start wizard and use it to create the configuration used for the test run.

c. In the Test Suite pane, choose the test suite that contains the tests that you you intend to run.

Use the `sampleJCK-runtime-6b` test suite for this example.

d. In the Work Directory pane, create a work directory for the test suite.

The work directory contains all of the information collected by the JavaTest harness during the test run of this test suite and configuration combination.

e. Select the Start the Configuration Editor checkbox and the Start test run checkbox.

The Quick Start wizard automatically opens the Configuration Editor for you when it closes. When you complete the configuration and close the Configuration Editor, the harness will automatically begin the test run.

f. Click the Finish button.

4 Create a configuration for the test run.

The JCK interview contains the following three major sections:

- The first section of each interview contains a series of sub-interviews that collect the information required to configure a test environment.
- The second section of the interview collects information about the tests from the JCK 6b test suite that is to be run.
- The last section of the interview collects information about how the tests are run (such as keywords and the exclude list) and contains JCK 6b specific information and values.

  a. In the first section of the interview, provide the answers from the More Information.

  b. Answer the remaining interview questions with settings required for your test system.

5 Click the Done button.

Because you selected the Start test run checkbox in the Quick Start wizard, the test run begins automatically when the Configuration Editor closes.

More Information Interview Responses for Runtime Test Run Remotely on a Windows System Using Multiple VMs

Interview Question	Answer
Test Execution Mode — Select the test execution mode	`MultiJVM`
Testing Locally — Are you running these tests on the same system used for this interview?	`No`
Test Platform — Which operating system is used on the test platform?	The appropriate Windows version

▼ Run JCK Runtime Tests Remotely on a Solaris System

1 Complete all required special test setup steps.

See "4.5 Special Setup Instructions" on page 104.

2 Start the JavaTest harness.

When starting the harness, you must allocate additional memory for opening the JCK test suite. Use the following command to start the harness from the directory where the `javatest.jar` file is located:

```
java -Xmx512m -XX:PermSize=256m -XX:MaxPermSize=256m -jar javatest.jar
```

Note – When allocating memory to the VM, do not exceed the actual amount of memory available on your system.

If you start the harness from a directory that does not contain the `javatest.jar` file, you must include its path in the command. For example:

```
java -Xmx512m -XX:PermSize=256m -XX:MaxPermSize=256m -jar /jck6b/sampleJCK-runtime-6b/lib/javatest.jar
```

In the example, the harness is started from the root directory. The `javatest.jar` file is located in the `jck6b/sampleJCK-runtime-6b/lib` directory. Change these values for your specific installation.

3 Use the Quick Start wizard to specify the test suite, work directory, and configuration loaded in the Test Manager window.

Note – If the harness opens the GUI without displaying the Quick Start wizard, launch the Quick Start wizard by choosing File -> Open Quick Start Wizard from the menu bar.

---

a. When the Quick Start wizard opens, click the Start a new test run radio button and the Next button.

b. Click the Create a new configuration radio button and the Next button.

If your group does not use configuration templates, you must use an empty JCK interview to create a configuration for the test run.

If your group uses configuration templates, you can load a template in the Quick Start wizard and use it to create the configuration used for the test run.

c. In the Test Suite pane, choose the test suite that contains the tests that you intend to run.

Use the `sampleJCK-runtime-6b` test suite for this example.

d. In the Work Directory pane, create a work directory for the test suite.

The work directory contains all of the information collected by the JavaTest harness during the test run of this test suite and configuration combination.

e. Select the Start the Configuration Editor checkbox and the Start test run checkbox.

The Quick Start wizard automatically opens the Configuration Editor for you when it closes. When you complete the configuration and close the Configuration Editor, the harness automatically begins the test run.

f. Click the Finish button.

4 Create a configuration for the test run.

The JCK interview contains the following three major sections:

- The first section of the JCK interview contains a series of sub-interviews that collect the information required to configure a test environment.
- The second section of the interview collects information about the tests from the JCK 6b test suite that is to be run.
- The last section of the interview collects information about how the tests are run (such as keywords and the exclude list) and contains JCK 6b specific information and values.

  a. In the first section of the interview, provide the answers from More Information.

  b. Answer the remaining interview questions with settings required for your test system.

5 Click the Done button.

Because you selected the Start test run checkbox in the Quick Start wizard, the test run begins automatically when the Configuration Editor closes.

More Information Interview Responses for Runtime Test Run Remotely on a Solaris System Using Multiple VMs

Interview Question	Answer
Test Execution Mode — Select the test execution mode	`MultiJVM`
Testing Locally — Are you running these tests on the same system used for this interview?	`No`
Test Platform — Which operating system is used on the test platform?	Solaris

B.2 Using a Browser to Run JCK Tests in a Single VM

The following procedures demonstrate how to set up and run JCK 6b tests with a browser (Netscape Navigator and JavaPlug-In software) and an agent for the following test environments and configurations:

- "Using a Single VM and an Active Agent to Run Tests on a Solaris System" on page 262
- "Using a Single VM and an Agent to Run Tests on a Windows System " on page 265

▼ Using a Single VM and an Active Agent to Run Tests on a Solaris System

1 Construct the applet code to launch the agent.

In the following example, create a file named `agentApplet.html` to contain the applet code. Example B–1 contains the applet code to launch an active agent. Refer to the *JavaTest Agent User's Guide* or the JavaTest harness online help for detailed instructions about creating and using map files, creating and using agent applets, and for starting the JavaTest agent.

Note – Some browser implementations support the `archive` tag. The archive tag supports multiple files. You must point to both the `javatest.jar` as well as the JCK classes. If your implementation only supports the `codebase` tag, you must unpack the `javatest.jar` into the same location as the JCK classes since the `codebase` tag only supports one single entry (`codebase`).

---

2 Start the agent applet by pointing your implementation at the `agentApplet.html` file on your web server.

Note – If you receive test execution errors, such as `Class not found`, you might be required to launch your browser implementation with a form of the `verify` switch.

---

3 Complete all required special test setup steps.

See "4.5 Special Setup Instructions" on page 104.

4 Start the JavaTest harness.

When starting the harness, you must allocate additional memory for opening the JCK test suite. Use the following command to start the harness from the directory where the `javatest.jar` file is located:

```
java -Xmx512m -XX:PermSize=256m -XX:MaxPermSize=256m -jar javatest.jar
```

Note – When allocating memory to the VM, do not exceed the actual amount of memory available on your system.

If you start the harness from a directory that does not contain the `javatest.jar` file, you must include its path in the command. For example:

```
java -Xmx512m -XX:PermSize=256m -XX:MaxPermSize=256m -jar /jck6b/sampleJCK-runtime-6b/lib/javatest.jar
```

In the example, the harness is started from the root directory. The `javatest.jar` file is located in the `jck6b/sampleJCK-runtime-6b/lib` directory. Change these values for your specific installation.

5 Use the Quick Start wizard to specify the test suite, work directory, and configuration loaded in the Test Manager window.

Note – If the harness opens the GUI without displaying the Quick Start wizard, launch the Quick Start wizard by choosing File -> Open Quick Start Wizard from the menu bar.

---

a. When the Quick Start wizard opens, click the Start a new test run radio button and the Next button.

b. Click the Create a new configuration radio button and the Next button.

If your group does not use configuration templates, you must use an empty JCK interview to create a configuration for the test run.

If your group uses configuration templates, you can load a template in the Quick Start wizard and use it to create the configuration used for the test run.

c. In the Test Suite pane, choose the test suite that contains the tests that you you intend to run.

Use the `sampleJCK-runtime-6b` test suite for this example.

d. In the Work Directory pane, create a work directory for the test suite.

The work directory contains all of the information collected by the JavaTest harness during the test run of this test suite and configuration combination.

e. Select the Start the Configuration Editor checkbox and the Start test run checkbox.

The Quick Start wizard automatically opens the Configuration Editor for you when it closes. When you complete the configuration and close the Configuration Editor, the harness automatically begins the test run.

f. Click the Finish button.

6 Create a configuration for the test run.

Each configuration interview contains the following three major sections:

- The first section of each interview contains a series of sub-interviews that collect the information required to configure a test environment.
- The second section of the interview collects information about the tests from the JCK 6b test suite that is to be run.
- The last section of the interview collects information about how the tests are run (such as keywords and the exclude list) and contains JCK 6b specific information and values.

  a. In the first section of the interview, provide the answers from the More Information table below.

  b. Answer the remaining interview questions with settings required for your test system.

7 Click the Done button.

Because you selected the Start test run checkbox in the Quick Start wizard, the test run begins automatically when the Configuration Editor closes.

Example B–1 Applet Code to Launch an Active JavaTest Agent

```
<applet
code="com.sun.javatest.agent.AgentApplet"
code="http://[INSERT-PATH-TO-JCK]/classes"
width=600
height=400>
<param name=mode value=active>
</applet>
```

More Information Interview Responses for Runtime Test Run on a Solaris System Using Single VM

Interview Question	Answer
Test Execution Mode — Select the test execution mode	`SingleJVM`
Test Platform File Separator — Which file separator character is used on the test platform?	`/`
Test Platform Path Separator — Which path separator character is used on the test platform?	:
Agent Type — You can run the JavaTest Agent in one of two modes: active or passive. Which mode do you wish to use?	active

▼ Using a Single VM and an Agent to Run Tests on a Windows System

1 Create a map file that translates host specific values into values that the agent can use.

Some tests require contextual information, such as the host name on which they are executed, before they can run. Because network file systems might be mounted differently on different systems, the path names used by the JavaTest harness might not be the same for the agent. The agent uses a map file to translate these strings into values it can use to run tests. Use a map file to map the work directory file syntax between systems. The map file is accessed by the agent. It might be networked or it might reside on the agent machine itself.

a. Use a text editor to open a simple ASCII file and enter the following types of lines:

i. Comment line - Begins with the pound sign (#) and provides information that is not processed by the agent. Comment lines are optional.

Example:

```
#Replace all /home/jjg with /jjg
```

ii. Translation line - Contains the target and substitution strings. Enter the string that is to be replaced followed by one or more spaces and the replacement string. The agent replaces all occurrences of the first string with the second.

Example:

```
/home/jjg /jjg
```

Because the agent uses the map file to perform global string substitution on all matching values received from the JavaTest harness, you must be as specific as possible when specifying strings in a translation line.

b. Save the map file using a name that is easily identifiable (such as `mapfile.txt`).

The mapping order (from left to right) in the map file is JavaTest harness to agent.

Example of a map file:

```
#This is a sample map file
#Replaces all /home/jjg with /jjg
/home/jjg /jjg
#Replaces all /home/kasmith/JavaTestharness with
#/kas/JavaTestharness
/home/kasmith/JavaTestharness /kas/JavaTestharness
```

2 Construct the applet code to launch the agent.

In the following example, create a file named `agentApplet.htmlto` contain the applet code. Example B–2 contains the applet code required to launch a passive agent. Example B–3 contains the applet code required to launch an active agent. Refer to the *JavaTest Agent User's*

Guide or the JavaTest harness online help for detailed instructions about creating and using agent applets, and for starting the JavaTest agent.

Note – Some browser implementations support the `archive` tag. The archive tag supports multiple files. You must point to both `javatest.jar` as well as the JCK classes. If your implementation only supports the `codebase` tag, you must unpack the `javatest.jar` into the same location as the JCK classes because the `codebase` tag only supports one single entry (`codebase`).

---

3 Start the agent applet by pointing your implementation at the `.html` file on your web server.

4 Define `PATH` for launching the JavaTest harness.

`PATH` must point to a reference JRE software VM.

5 Complete all required special test setup steps.

See "4.5 Special Setup Instructions" on page 104.

6 Start the JavaTest harness.

When starting the harness, you must allocate additional memory for opening the JCK test suite. Use the following command to start the harness from the directory where the `javatest.jar` file is located:

```
c:\>java -Xmx512m -XX:PermSize=128m -XX:MaxPermSize=128m -jar javatest.jar
```

Note – When allocating memory to the VM, do not exceed the actual amount of memory available on your system.

If you start the harness from a directory that does not contain the `javatest.jar` file, you must include its path in the command. For example:

```
c:\>java -Xmx512m -XX:PermSize=128m -XX:MaxPermSize=128m -jar c:\jck\sampleJCK-runtime-6b\lib\javatest.jar
```

In the example, the harness is started from the root directory of the `c:` drive. The `javatest.jar` file is located on the `c:` drive in the *jck*`\sampleJCK-runtime-60\lib` directory. Change these values for your specific installation.

7 Use the Quick Start wizard to specify the test suite, work directory, and configuration loaded in the Test Manager window.

Note – If the harness opens the GUI without displaying the Quick Start wizard, launch the Quick Start wizard by choosing File -> Open Quick Start Wizard from the menu bar.

---

a. When the Quick Start wizard opens, click the Start a new test run radio button and the Next button.

b. Click the Create a new configuration radio button and the Next button.

If your group does not use configuration templates, you must use an empty JCK interview to create a configuration for the test run.

If your group uses configuration templates, you can load a template in the Quick Start wizard and use it to create the configuration used for the test run.

c. In the Test Suite pane, choose the test suite that contains the tests that you you intend to.

Use the `sampleJCK-runtime-6b` test suite for this example.

d. In the Work Directory pane, create a work directory for the test suite.

The work directory contains all of the information collected by the JavaTest harness during the test run of this test suite and configuration combination.

e. Select the Start the Configuration Editor checkbox and the Start test run checkbox.

The Quick Start wizard automatically opens the Configuration Editor for you when it closes. When you complete the configuration and close the Configuration Editor, the harness automatically begins the test run.

f. Click the Finish button.

8 Create a configuration for the test run.

Each configuration interview contains the following three major sections:

- The first section of each interview contains a series of sub-interviews that collect the information required to configure a test environment.
- The second section of the interview collects information about the tests from the JCK 6b test suite that is to be run.
- The last section of the interview collects information about how the tests are run (such as keywords and the exclude list) and contains JCK 6b specific information and values.

  a. In the first section of the interview, provide the answers from the More Information table below.

  b. Answer the remaining interview questions with settings required for your test system.

9 Click the Done button.

Because you selected the Start test run checkbox in the Quick Start wizard, the test run begins automatically when the Configuration Editor closes.

Example B–2 Applet Code to Launch a Passive JavaTest Agent

```
<applet
code=com.sun.javatest.agent.AgentApplet
codebase=url-to-jck-root/classes
archive=url-to-jck-root/lib/javatest.jar
width=600
height=600>
param name=mode value=passive>
<param name=map value=mapfile.txt>
</applet>
```

Example B–3 Applet Code to Launch an Active JavaTest Agent

```
<applet
code=com.sun.javatest.agent.AgentApplet
codebase=url-to-jck-root/classes
archive=url-to-jck-root/lib/javatest.jar
width=600
height=600>
<param name=mode value=active>
<param name=map value=mapfile.txt>
<param name=activeHost value=[name-of-javatest-machine]>
<param name=activePort value=[javatest-agentpool-port]>
</applet>
```

More Information Interview Responses for Runtime Test Run on a Windows System Using Single VM

Interview Question	Answer
Test Execution Mode — Select the test execution mode	`SingleJVM`
Test Platform File Separator — Which file separator character is used on the test platform?	\
Test Platform Path Separator — Which path separator character is used on the test platform?	;
Agent Type — You can run the JavaTest Agent in one of two modes: active or passive. Which mode do you wish to use?	active

◆ ◆ ◆ APPENDIX C

Kerberos Key Distribution Center Example Setup

The Kerberos Key Distribution Center (KDC) can be set up on the same or remote machine as long as the system under test is networked to it.

C.1 Setup

The following is an example of the setup process for a Sun Microsystems machine running the Solaris platform, version 2.8.

Note – If your network already has a KDC, skip to "To Add Kerberos Principals for JCK Testing" on page 275.

The setup process consists of the following tasks:

C.1.1 Obtaining the Kerberos Software

You can obtain the Kerberos software on CD-ROM or by download from Sun as well as from other vendors.

C.1.1.1 Kerberos Software on CD-ROM

You can order the software CD-ROM for the Solaris platform, version 2.8, from Sun Microsystems, Inc. with this title and part number:

Solaris 8 Admin Pack CD-ROM
January 2000, Revision A
Part #704-7142-10

C.1.1.2 Kerberos Software by Download

You can download the software for free from the following web site:

`http://www.sun.com/bigadmin/content/adminPack/index.html`

You do not have to have a sign-on account to download the Kerberos software from this site.

After you log in and agree to the license agreement, you are presented with multiple options from the download list. Choose the following option:

Download Software CD Image, Solaris/Intel, Solaris/SPARC, Windows/Intel
(80.20 MB)

C.1.1.3 Kerberos Software From Other Vendors

You can also use comparable products from other software vendors and distributors. However, Sun has not installed or tested these products. If you are unable to set up the KDC and run security tests using products from other software vendors and distributors, obtain the Kerberos software from Sun either on CD-ROM or by download.

C.1.2 Installing Kerberos Software on a Solaris Platform

The following is an example of the installation process for a Sun Microsystems machine running Solaris platform, version 2.8. If your network already has a KDC, skip to "C.1.3 Adding Kerberos Principals for JCK Testing" on page 275.

Caution – The KDC machine's clock must be set within five minutes of the test machine clock.

▼ To Install Kerberos Software on a Solaris Platform

1 **In the directory where you have saved the downloaded binary, unzip the `Solaris_8_Admin_Pack.zip` file and run the installation script as follows:**

```
%unzip Solaris_8_Admin_Pack.zip
%chmod +x installer
%installer
```

This requires the root password of your machine. The installation script brings up the wizard to guide you through the installation.

Note – The following figures illustrate the basic steps required to set up an example Kerberos realm called PLOP, on a machine called `plop.sfbay.sun.com`, in an example network domain name `jlaps.sfbay.sun.com`.

2 **Click Next at the bottom of the Welcome dialog, Figure C–1.**

FIGURE C–1 Welcome Screen - Solaris 8 Admin Pack

3 **For this installation, choose Custom Install on the Select Type of Install dialog box, Figure C–2.**

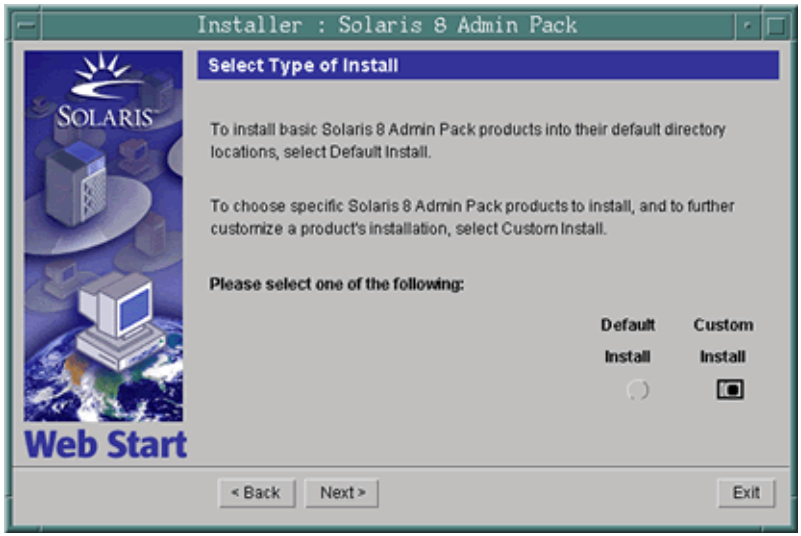


FIGURE C–2 Select Type of Install

4 **Click Next to continue.**

5 **For this installation, choose Sun Enterprise Authentication Mechanism on the Product Selection dialog box, Figure C–3.**

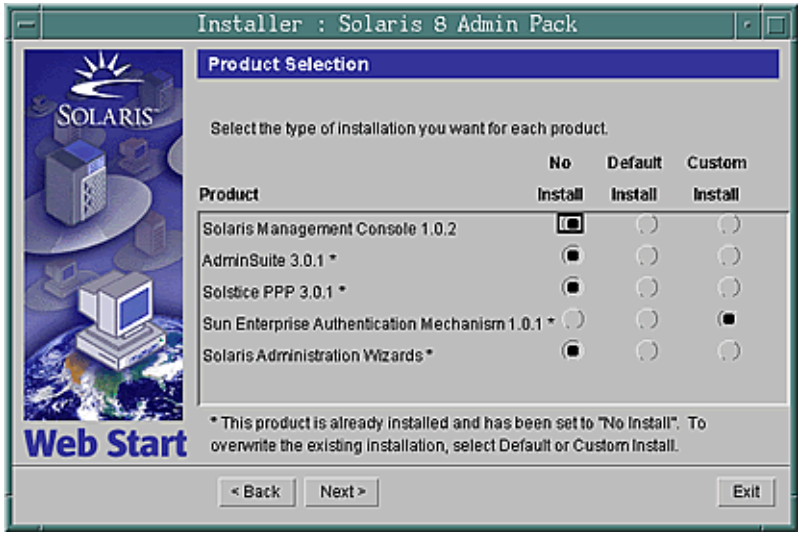


FIGURE C–3 Product Selection

Note – Additional products can be installed, but the Sun Enterprise Authentication Mechanism is the minimum installation required to set up a KDC.

6 **Click Next to continue.**

7 **For this installation, choose SEAM Master KDC on the Component Selection dialog box, Figure C–4.**

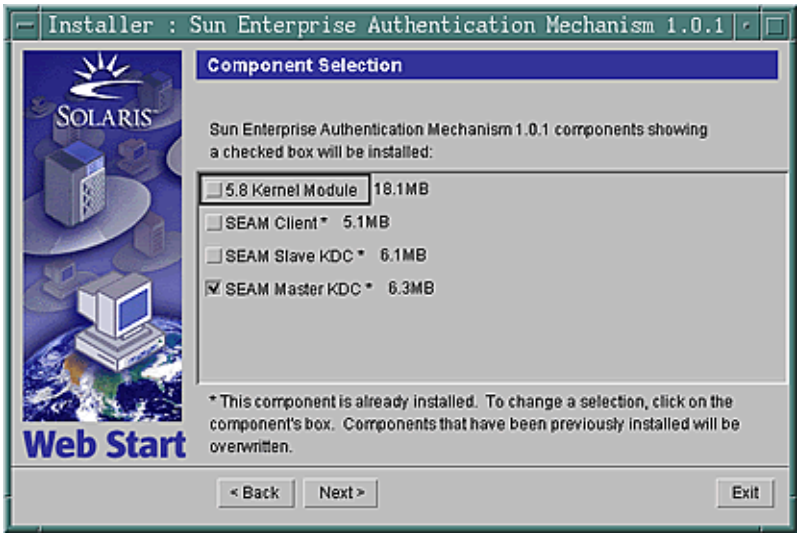


FIGURE C–4 Component Selection

8 **Click Next to continue.**

9 **Configure the site for the KDC by using the buttons and text fields on the Site Configuration dialog box, Figure C–5.**

Note – You might want to consult your system administrator before completing the Site Configuration screen.

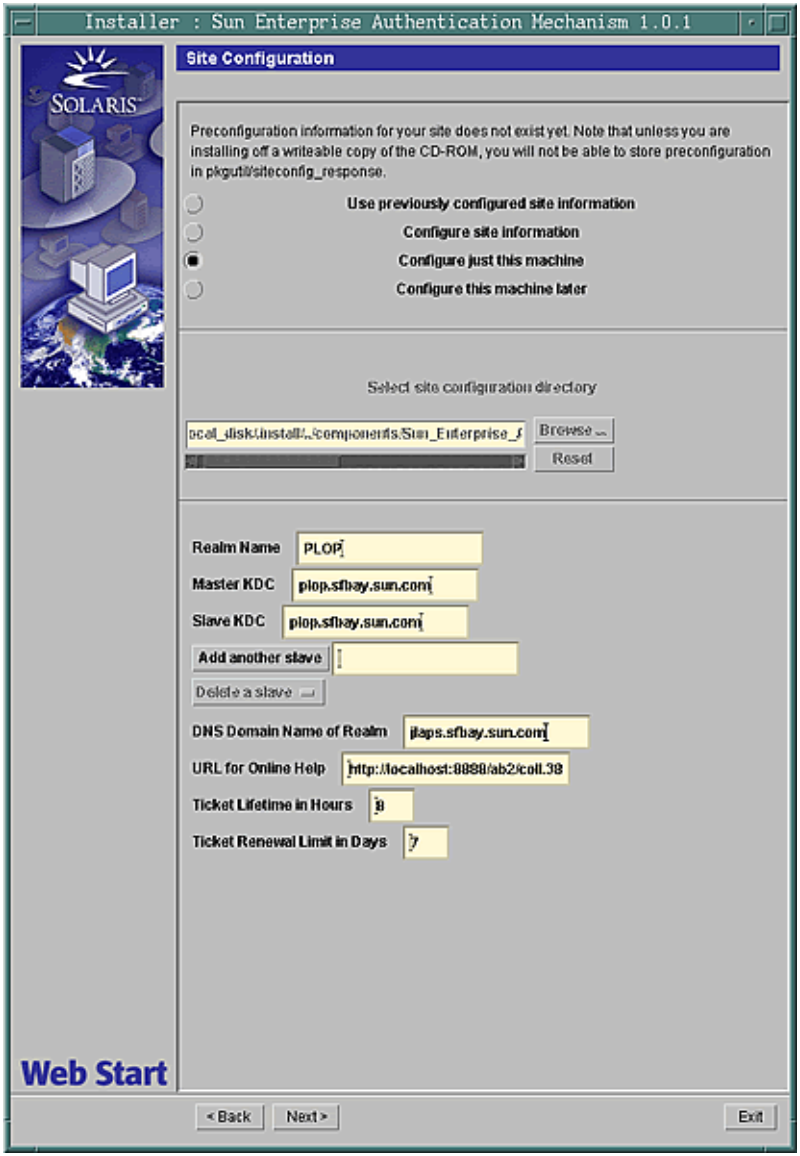


FIGURE C–5 Site Configuration

The following Site Configuration settings are required for JCK testing:

- **Realm Name** - The Kerberos realm of the configured machine (such as TESTING, EXAMPLE, or JCK).

- **Master KDC** - The fully qualified name of the host or master KDC. In this example, Master KDC is `plop.sfbay.sun.com` and the name of the machine is `plop`.
- **Slave KDC** - Not required.
- **DNS Domain Name of Realm** - The domain name of the configured machine.
- **URL for Online Help** - URL for the online help page. Use the default value.
- **Ticket Lifetime in Hours** - The maximum life of the ticket before it must be renewed. Use the default value of 8 hours.
- **Ticket Renewal Limit in Days** - The maximum number of days that a ticket can be renewed. Use the default value of 7 days.

10 **Click the Exit button after the site configuration settings are completed.**

11 **Reboot your machine.**

C.1.3 Adding Kerberos Principals for JCK Testing

Note – For current detailed information regarding Kerberos, including Sun Enterprise Authentication Mechanism Guide (SEAM), open web page `http://docs.sun.com` and search for the keyword `kerberos`. A list of documents is displayed that you can refer to for current information.

▼ To Add Kerberos Principals for JCK Testing

1 **Verify the krb5.conf file is installed.**

See Example C–1 for an example of the `krb5.conf` file.

2 **Verify the `kdc.conf` file is installed.**

See Example C–2 for an example of the `kdc.conf` file.

3 **Create the KDC Database using `kdb5_util`.**

a. **Enter the following command:**

```
%/usr/krb5/sbin/kdb5_util create -r PLOP -s
```

The system displays the following:

```
Initializing database '/var/krb5/principal' for realm 'PLOP'
master key name 'PLOP@PLOP'
Enter KDC database master key: <type the key>
```

b. At the prompt, enter KDC database master key.

The system displays the following:

```
Re-enter KDC database master key to verify: <type it again>
```

c. At the prompt, re-enter the KDC database master key.

4 Create Kerberos principals using `kadmin.local`.

a. Enter the following command:

```
%/usr/krb5/sbin/kadmin.local
```

The system displays the following:

```
kadmin.local: ank user1
```

b. At the prompt, enter the password for `principal` `user1@PLOP`.

c. Record the password for later use.

d. At the prompt, re-enter the password for `principal` `user1@PLOP`.

The system displays the following:

```
Principal "user1@PLOP" created.
kadmin.local: ank user2
Enter password for principal user2@PLOP: <type the password>
```

e. At the prompt, enter the password for `principal` `user2@PLOP`.

The system displays the following:

```
Re-enter password for principal user2@PLOP: <type it again>
```

f. At the prompt, re-enter password for `principal` `user2@PLOP`.

The system displays the following:

```
Principal "user2@PLOP" created.
```

Example C–1 Sample `krb5.conf` File

```
%cat /etc/krb5/krb5.conf
# Copyright (c) 1998, by Sun Microsystems, Inc.
# All rights reserved.
#
#pragma ident "@(#)krb5.conf 1.10 98/11/11 SMI"
[libdefaults]
default_realm = PLOP
[realms]
PLOP = {
```

```
kdc = plop.sfbay.sun.com
kdc = plop.sfbay.sun.com
admin_server = plop.sfbay.sun.com
}
[domain_realm]
.jlaps.sfbay.sun.com = PLOP
...
```

Example C–2 Sample `kdc.conf` File

```
%cat /etc/krb5/kdc.conf
[kdcdefaults]
kdc_ports = 88,750
[realms]
PLOP = {
profile = /etc/krb5/krb5.conf
database_name = /var/krb5/principal
admin_keytab = /var/krb5/kadm5.keytab
acl_file = /var/krb5/kadm5.acl
kadmind_port = 749
max_life = 8h 0m 0s
max_renewable_life = 7d 0h 0m 0s
}
```

◆ ◆ ◆ APPENDIX D

JavaTest Harness Tutorial

This tutorial introduces the JavaTest harness GUI and some basic underlying concepts. The tutorial instructs you to run a small test suite called `demotck`. The `demotck` test suite contains 17 tests that test the functionality of some very simple demo APIs.

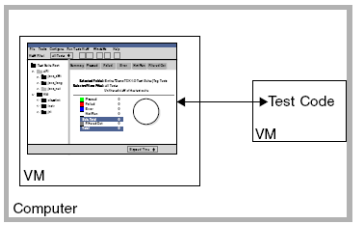
This tutorial describes how to do the following steps:

Use version 5.0 (or later) of the Sun Java SE platform on either the Microsoft Windows (Win32) or Solaris Operating System.

Note – Unless otherwise indicated, all examples in this book use Microsoft Windows-style command prompts and file separators.

D.1 Starting the JavaTest Harness

To keep things simple, these instructions show you how to run both the JavaTest harness and the tests on the same system in different VMs (see Figure D–1).



The JavaTest harness and the tests run on separate VMs and each test runs in a new VM.

FIGURE D–1 JavaTest Harness and Tests Running on Same System

It is also possible to run the JavaTest harness and the tests on separate systems using the JavaTest harness agent.

▼ To Start the JavaTest Harness

1 Verify that Java SE 6 software is in your path.

At a command prompt, enter:

```
C:\> java -version
```

2 Make <*jck*>`\doc\javatest\tutorial\simpleHTML\demotck` **the current directory.**

Where *jck* is the JCK installation directory.

Note – Run the JavaTest harness on a computer with a minimum of 256 megabytes of physical memory. Use an agent with the JavaTest harness to run test programs on platforms with limited amounts of memory. If you are running a large test suite (10,000 or more tests), you must allocate a large amount of memory to the VM to run the JavaTest harness properly. Allocate 512 megabytes when running a large test suite by adding `-Xmx512m` to the command string when you start the JavaTest harness from a writable directory. When allocating memory to the VM, do not exceed the actual amount of memory available on your system.

---

3 Start the JavaTest harness.

On UNIX systems, enter the following command at the command prompt:

```
java -Xmx512m -jar lib/javatest.jar -newDesktop
```

On Win32 systems, enter the following command at the command prompt:

```
java -Xmx512m -jar lib\javatest.jar -newDesktop
```

Note – The `-newDesktop` option is used here to ensure that the JavaTest harness starts exactly as described in these instructions. Under normal circumstances do *not* use this option. When you use this option, you lose any information that the harness saved about your previous session. For information about JavaTest harness options, see the JavaTest harness online help.

The JavaTest harness starts and displays the Quick Start wizard window as shown in Figure D–2.

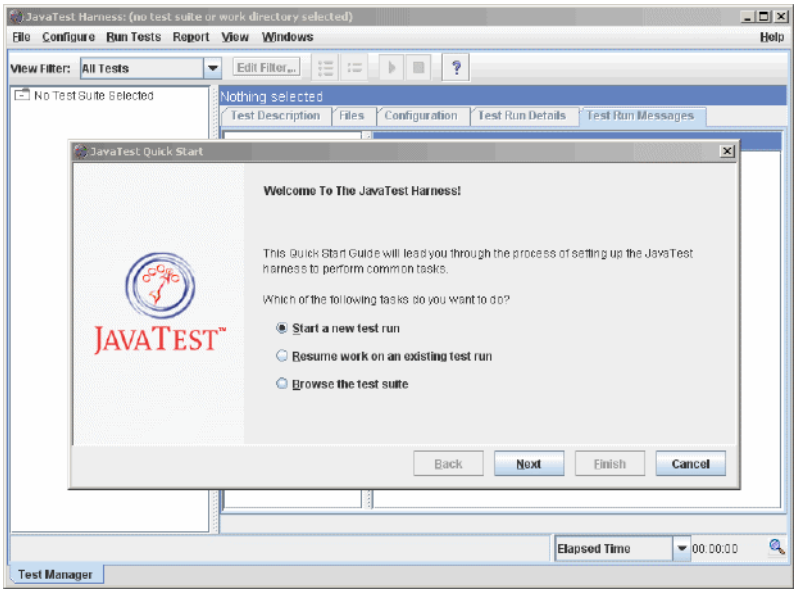


FIGURE D–2 The JavaTest Harness With Quick Start Wizard

D.2 Using the Quick Start Wizard

The Quick Start wizard leads you through the basic steps required to start running the test suite.

▼ To Use the Quick Start Wizard

1. **Choose Start a new test run on the Welcome to the JavaTest Harness screen. Click Next**
2. **Choose Create a new configuration on the Configuration screen. Click Next.**
3. **Click Next on the Test Suite screen to accept the default test suite setting.**

4 **Click Browse on the Work Directory screen. Click Next.**

The JavaTest harness uses the work directory to store information and to write test results. Use the file chooser to create the work directory in a convenient location *outside of the test suite directory* (`demotck`).

5 **Click Finish on the Almost Done screen to accept the default setting and complete the Quick Start process.**

The default setting automatically starts the Configuration Editor when the Quick Start wizard closes.

D.3 Configuring Test Information

Because you used the default setting in the last panel of the Quick Start wizard, the Configuration Editor starts automatically.

Use the Configuration Editor to configure the information required to run the test suite. The Configuration Editor consists of three panes and a menu bar, as shown in Figure D–3.

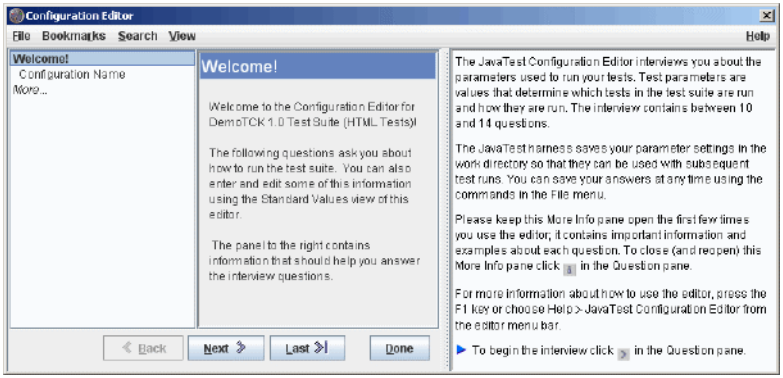


FIGURE D–3 JavaTest Harness Configuration Editor Example

The left pane lists the titles of the questions you have answered, are currently answering, or that the editor deduces must be answered. The current question is highlighted.

The center pane displays the interview questions. Answer the questions by using controls such as text boxes, radio buttons, or drop-down menus located below each question. Whenever possible, the editor deduces answers from your system configuration and includes them in text boxes, combo boxes, and radio buttons. You can accept these answers or provide other answers.

The right pane displays important information about each question, including background information, examples of answers, and additional information about choosing an answer.

- Answer the questions displayed in the configuration editor.

  The Demo interview contains 13 questions. Table D–1 presents the titles, answers, and information about each question that you must answer in the interview.

TABLE D–1 Tutorial Interview Questions and Answers

Question Title	Answer	Description
Welcome!		Briefly describes the purpose and function of the Demo Configuration Editor.
Configuration Name	Demo	Names the interview file.
Description	tutorial	Describes the configuration.
How to Run Tests	On this computer	Runs both the JavaTest harness and the tests on the same computer.
Java Virtual Machine	The absolute path to the `java` command on a win32 system. For example: *jdk_inst_dir*`\bin\java` or *jre_inst_dir*`\jre\java`	Click Browse to activate a file chooser, or type the path directly in the text box.
Test Verboseness	Medium	Causes all executing tests to emit standard information messages.
Parameters...		Introduces the section of questions about the tests to run and how to run them.
Specify Tests to Run?	No	Runs all of the tests.
Specify an Exclude List?	No	Specifies that an exclude list is not used for this test run.
Specify Status?	No	Specifies that prior run status is not used to filter the test run. Feel free to try it on subsequent runs.
Concurrency	1	Specifies the default concurrency setting (1).
Time Factor	1	Specifies the default standard time out value for each test (1).

TABLE D–1 Tutorial Interview Questions and Answers *(Continued)*

Question Title	Answer	Description
Congratulations!		The configuration editor has all of the information it needs to run the tests. Click the Done button to save the interview. JavaTest harness interviews are saved to files that end with the `.jti` suffix. Use the file chooser to specify a file in a convenient location.

D.4 Running Tests

▼ To Run the Tests

1 Change the View Filter setting in the tool bar from All Tests to Last Test Run.

Choose Last Test Run in the View Filter drop-down menu located in the tool bar. See Figure D–4, for the location of the View Filter combo box. This changes your view of the test tree so that you only see the results of the current test run. This is generally the view that most users prefer. Refer to the *JavaTest Harness User's Guide: Graphical User Interface* for a detailed description of the JavaTest harness view filters.

Note – When you change to the Last Run filter before you do a test run, the folders and tests in the tree turn to gray, indicating that they are filtered out. This occurs because results fo not exist from your last test run.

---

2 Choose Run Tests -> Start to start the test run.

The harness begins to run tests. You can see activity in the test tree panel that indicates the tests currently running. You can also watch the progress of the test run in the progress monitor on the bottom-right portion of the JavaTest harness window and the pie chart in the Summary tab.

3 Expand the test tree folders to reveal the tests.

Click different test folders to expand the test tree. See Figure D–4 for an example of an expanded test tree.

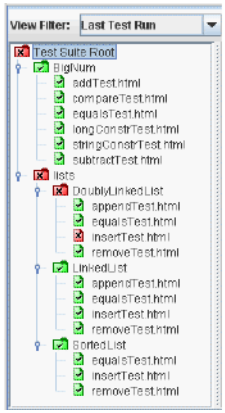


FIGURE D–4 Expanded Test Tree Example

D.4.1 JavaTest Harness GUI — Folder and Test Colors

As tests complete, the tests and their folders change color to represent their state.

Table D–2 briefly describes the colors and their meaning.

TABLE D–2 Folder and Test Colors and Their Meaning

Color	Description
Green	Passed.
Red	Failed.
Blue	Error. The test could not be run properly. Usually indicates a configuration problem.
Gray	Filtered out. Due to a parameter setting (for example, it is on an exclude list), the test is not selected to be run.
White	Not run.

Folders reflect the state of the tests hierarchically beneath them. You know that the entire test suite passed if the test suite root folder is green. See the JavaTest harness online help for more information.

Note – The test `lists\DoublyLinkedList\InsertTest.html` intentionally contains errors and is supposed to fail as part of the tutorial. If any other tests fail, check your answers to the configuration interview.

D.5 Browsing Test Results

When the test run is complete, use the Folder tabbed pane and Test tabbed pane portion of the JavaTest harness to examine the results. Examine the output of the test that failed.

Note – The Folder tabbed pane and the Test tabbed pane occupy the same portion of the Test Manager window. The Folder tabbed pane is displayed when you choose a folder entry in the test tree and the Test tabbed pane is displayed when you choose a test entry in the test tree.

D.5.1 The Folder Pane

The Folder tabbed pane displays information about the tests in the selected folder. Figure D–5 shows an example of the folder pane.

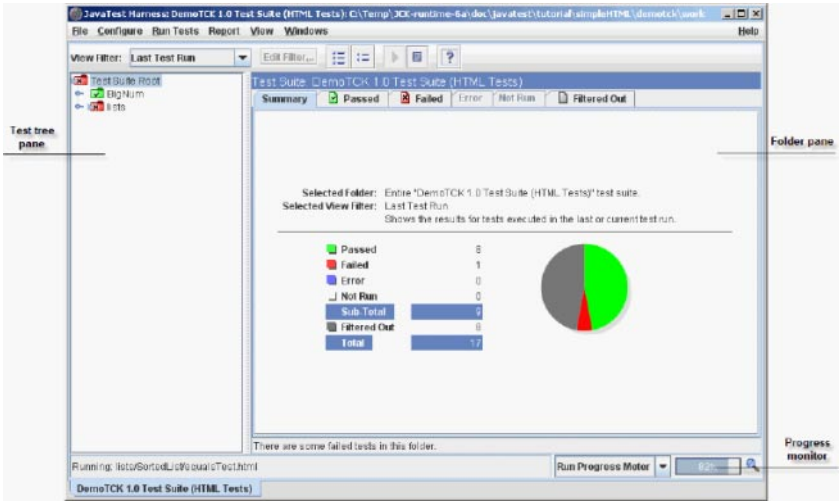


FIGURE D–5 Folder Pane Example

▼ Using the Folder Pane

1. **Click the top folder in the test tree (the test tree root).**

2. **Click the Summary tab (shown by default) to display the Summary panel.**

   Notice the statistics displayed in the Summary panel. It describes how many tests in the test suite passed, failed, had errors, and were filtered out.

3. **Click any of the other folder icons in the test tree to display the statistics for the selected tests.**

   This action causes the Summary panel to display the statistics for all tests hierarchically beneath the selected folder.

4. **Click the test tree root folder again to redisplay the test suite statistics.**

   This action causes the Summary panel to redisplay the statistics for the complete test suite.

5. **Click the Passed tab to display a list of tests that passed during the test run.**

6. **Click the Failed tab to display a list of the tests that failed during the test run (only one test in this case).**

7. **Double-click the `lists\DoublyLinkedList\InsertTest.html` test in the Failed tab to change the display from the Folder pane to the Test pane.**

   This action selects the test in the test tree and changes the display from the Folder pane to the Test pane.

Note – To read more information about any of the panes, click on a tab to establish focus and press F1 to activate online help about that pane.

D.5.2 The Test Pane

The Test tabbed pane displays information about the selected test. The five tabs provide information about the test and information about the results of its execution. Figure D–6 shows an example of the test pane.

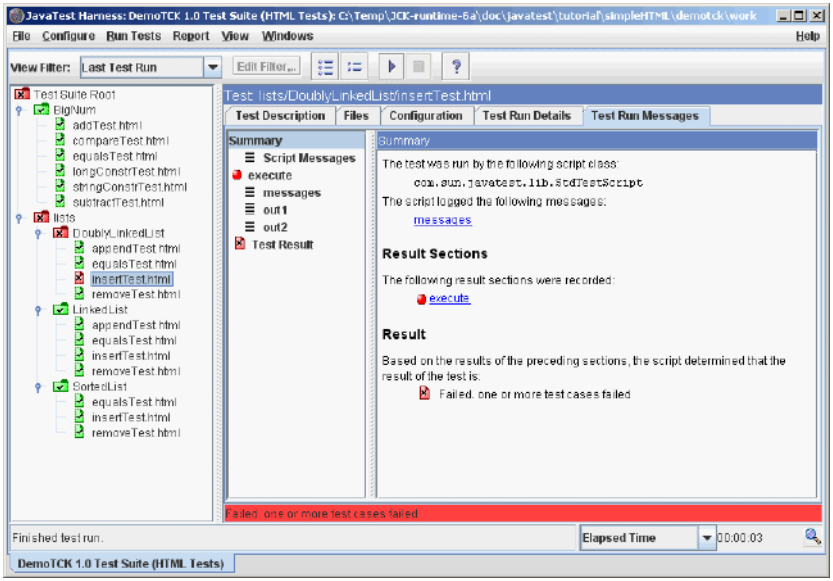


FIGURE D–6 Test Pane Example

Each tab displays information collected by the JavaTest harness about the test highlighted in the test tree.

Table D–3 briefly describes each tabbed pane.

TABLE D–3 Test Pane Tabs

Tab	Description
Test Run Messages	Displays messages generated during the selected test's execution
Test Run Details	A table of values generated during the selected test's execution
Configuration	A table of the configuration values used during the selected test's execution
Files	Displays the source code and any other files related to the selected test
Test Description	A table of the test description values specified for the test

Note – To read more information about any of the panes, click on a tab to establish focus, and press F1 to activate the online help about that pane.

▼ Using the Test Pane

1 **Click the Test Run Messages tab to view the messages generated by the JavaTest harness or the test during execution.**

Notice that the red icons indicate that the test failed.

2 **Click the execute messages item in the left column to display the command line used to run the test.**

The display on the right shows the command line used to run the test. Problems can often be debugged by examining how the test was invoked. In this case it was invoked correctly. See Figure D–7 for an example of the test messages pane.

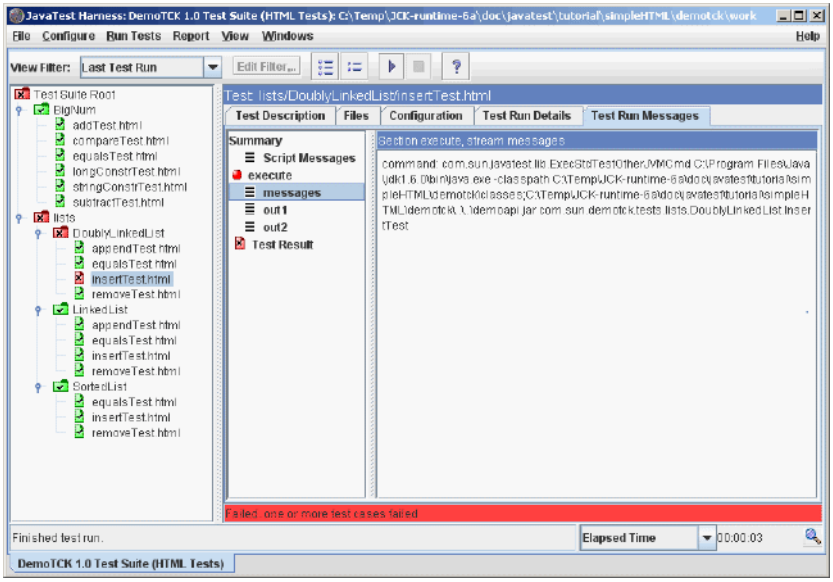


FIGURE D–7 Test Messages Pane Example

3 **Click the out1 item in the left column to display errors reported by the test.**

The display on the right shows errors reported by the test. The messages indicate that either the test or the API contain errors. In this example, the test contains errors. Figure D–8 shows an example of logged error messages in the test messages pane.

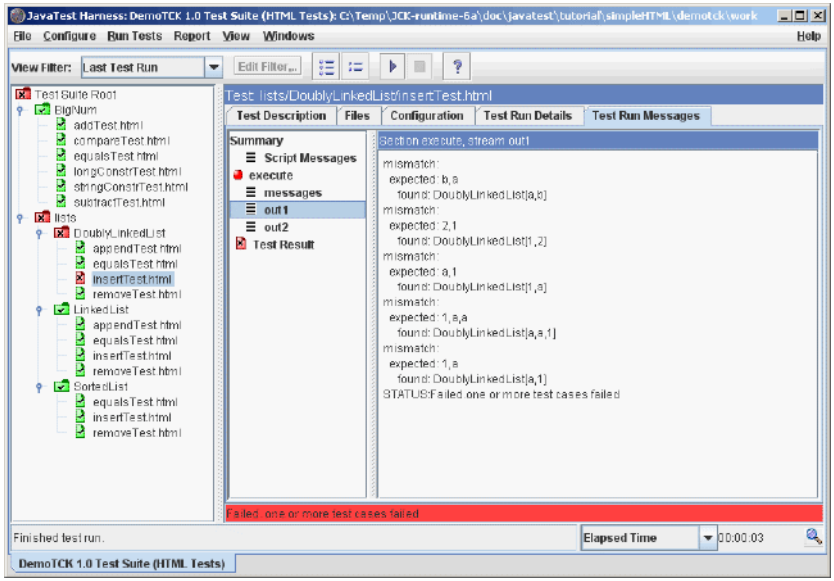


FIGURE D–8 Logged Error Messages Example

D.6 Excluding a Failed Test

The JavaTest harness allows you to exclude tests from a test suite by specifying an *exclude list* file. This section shows you how to use the Quick Set Mode of the Configuration Editor to specify an exclude list that includes `lists\DoublyLinkedList\InsertTest.html`. Tests that are excluded are not executed during test runs, and though they are still displayed in the test tree, their status is not reflected in the pass-fail status of the test suite.

▼ To Exclude a Failed Test

1. **Choose Configure -> Change Configuration -> Exclude List from the Test Manager menu bar.**

   The Configuration Editor opens directly to a panel that enables you to specify an exclude list. Quick Set Mode allows you to quickly change values that change frequently between test runs. These values are also referred to as *standard values*. Standard values can also be changed using the Configuration Editor in Question Mode.

2. **In the Exclude List pane, click Other.**

   This activates a tool used to specify a set of exclude lists.

3. **Click the Add button on the upper-right portion of the tool.**

   This invokes a file chooser used to specify an exclude list. The current directory of the file chooser is the directory that you were in when starting the JavaTest harness. If it is not, navigate to that directory.

4. **Double-click the `lib` directory entry in the file chooser.**

5. **Choose the `demo.jtx` entry in the file chooser and click Select.**

   Notice that the exclude list (`demo.jtx`) is added to the Exclude Lists text box.

6. **Click Done in the Configuration Editor.**

7. **Change the View Filter setting in the tool bar from Last Test Run to Current Configuration.**

   Choose Current Configuration in the View Filter located in the tool bar. Figure D–4 displays the location of the View Filter. This changes your view of the test tree so that it displays only the results of the tests selected and filtered out in the configuration. This filter shows only the tests that will run next, as opposed to the Last Test Run filter that shows only the tests that have been run.

   Notice that the icon for the `InsertTest.html` entry in the Test tree changes from red to gray. This indicates that the test is filtered out and will not be executed. Also notice that the Test Suite Root folder changes from red to green, indicating that all the currently selected tests passed.

D.7 Generating a Report

You can use the JavaTest harness to generate an HTML report that describes the results of the test run. All of the information contained in the report is available from the GUI. The following steps describe how to generate and browse a report of the test run completed in the previous sections of this tutorial.

▼ To Generate a Report

1. **Choose Report -> Create Report.**

   The Create a New Report dialog box opens.

2. **Specify the directory where you want the JavaTest harness to write the report files.**

   Use a file chooser to specify the directory and click on the Browse button.

3. **Click the Create Report(s) button.**

   The reports are generated and you are asked whether you want to view the report.

4 **Click Yes.**

The reports are displayed in the JavaTest harness report browser window. Scroll through the report and follow the various links to view data about the test run.

5 **To print the report, open the report in your web browser and print it.**

◆ ◆ ◆ APPENDIX E

Running a Single Test From Command Line

The JCK tests can be run from command line, which can be helpful for debugging purposes.

The JCK-runtime, JCK-compiler, and JCK-devtools test suites include sample UNIX shell scripts that show you how the tests are executed. To install shell scripts along with JCK, specify `-install shell_scripts` option during the installation. After the installation, the scripts are located in the test source directories - one script per test description. The purpose of the sample scripts is to show what information is required to run a test in a debugger or from a command line:

- what value should classpath contain
- which class should be run
- what arguments need to be passed

All configuration information is stored in the .html files in TestDescription tables. JavaTest Harness reads TestDescription and runs the test accordingly. When a test requires a configuration parameter, JavaTest takes its value from interview answers. Provided scripts show how the JavaTest Harness runs the test. For most configuration parameters, the typical value that suits most configurations is used. For other parameters, like network configuration where values can vary from system to system, the values must be set prior to script execution.

E.1 Running Scripts

Scripts can be run with a shell interpreter that is compatible with UNIX Korn shell. Scripts can be also run on a Windows system with MKS toolkit. In examples below 'sh' is used as shell interpreter. On some platforms 'ksh' should be used instead. Note that you need to set JAVA_HOME environment variable pointing to Java Implementation under test prior to script execution.

EXAMPLE E–1 Setting JAVA_HOME variable

```
# export JAVA_HOME=/opt/java6
```

EXAMPLE E–2 Running scripts from a Command Line

sh tests/api/java_lang/Package/index_Package.ksh

or

cd tests/api/java_lang/Package

sh index_Package.ksh

EXAMPLE E–3 Running scripts located outside the JCK directory

When script modification is required, the script file can be copied to another location, modified and run from there. To run a script located outside of the JCK install directory, set the TCK_HOME environment variable to point to JCK location.

cp /jck/JCK-runtime-6b/tests/api/java_lang/Package/index_Package.ksh /tmp

export TCK_HOME=/jck/JCK-runtime-6b

sh /tmp/index_Package.ksh

EXAMPLE E–4 Running scripts that use default working directory

Some tests create temporary files in the work directory. /tmp/jck-workdir is used as the default work directory. The directory is created automatically and after script execution, contains files created by tests.

sh tests/api/java_io/FileWriter/index_FWFlushTest.ksh

/tmp/jck-workdir is created

EXAMPLE E–5 Running scripts with specified working directory

Work directory can be defined by setting TESTWORKDIR variable.

export TESTWORKDIR=/tmp/FileWriter

sh tests/api/java_io/FileWriter/index_FWFlushTest.ksh

/tmp/FileWriter is created

EXAMPLE E–6 Running scripts with configuration parameters

There are scripts that require extra configuration parameters. These parameters should be set in the environment variables.

export NETWORK_LOCALHOST=myhost/123.10.11.12

EXAMPLE E–6 Running scripts with configuration parameters *(Continued)*

```
# export NETWORK_TESTHOST1=neighbour.domain/10.11.12.13

# sh tests/api/java_net/InetAddress/index_ToString.ksh
```

E.1.1 Hints for Running Scripts on a Windows System

Note – Make sure MKS toolkit is installed before running the scripts.

- For all tests

  Before running the script make sure that an environment variable that specifies the classpath, like CLASSPATH, ClassPath, etc is not set in the system. You can use the following commands to unset the selected environment variable:

  ```
  #> env | grep -i classpath

  #> unset ClassPath
  ```

- For those tests that require a workdir to create temporary files, like compiler or devtools tests:

  By default `/tmp/jck-workdir` directory is used for creating temporary files. If there is a problem creating such directory, TESTWORKDIR environment variable should to be set prior test script execution as follows:

  ```
  #> set TESTWORKDIR=C:/tmp
  ```

  or

  ```
  #> set TESTWORKDIR=C:\\tmp
  ```

- For devtools tests

  All devtools tests use one of four wrapper scripts to run schema compiler, schema generator, wsimport or wsgen tool. JCK-devtools test suite provides shell scripts for Solaris and Linux, and bat scripts for Windows. Solaris versions are used by default. To be able to run the test execution scripts under Windows platform, the similar shell scripts need to be provided, and path to them should be set via environment variables as shown below. SHELL variable should also be set:

  ```
  #> set SHELL=sh
  #> set TESTWORKDIR=C:/tmp
  #> set WSIMPORT=C:/jck-devtools/win-wsimport.sh
  #> set WSGEN=C:/jck-devtools/win-wsgen.sh
  #> set XJC=C:/jck-devtools/win-xjc.sh
  #> set SCHEMAGEN=C:/jck-devtools/win-schemagen.sh
  ```

Refer to "5.11 JAX-WS Mapping Tests" on page 136, "5.29 Schema Compiler" on page 202, and "5.30 Schema Generator" on page 205 chapters for details on the wrapper scripts.

◆ ◆ ◆ APPENDIX F

JCK Tools

This chapter contains information about programs, which are installed with JCK, but can be run separately to perform various JCK—related tasks.

F.1 Quick Configuration Editor

Quick Configuration Editor (QCE) is a standalone utility that helps you to configure JCK via a dialog box with a number of screens that contain properties for editing. You can use QCE to edit specific properties in the selected configuration.

Note – Quick Configuration Editor is an early access feature that is currently under development. Sun encourages you to use QCE for evaluation purposes only.

F.1.1 Starting Quick Configuration Editor

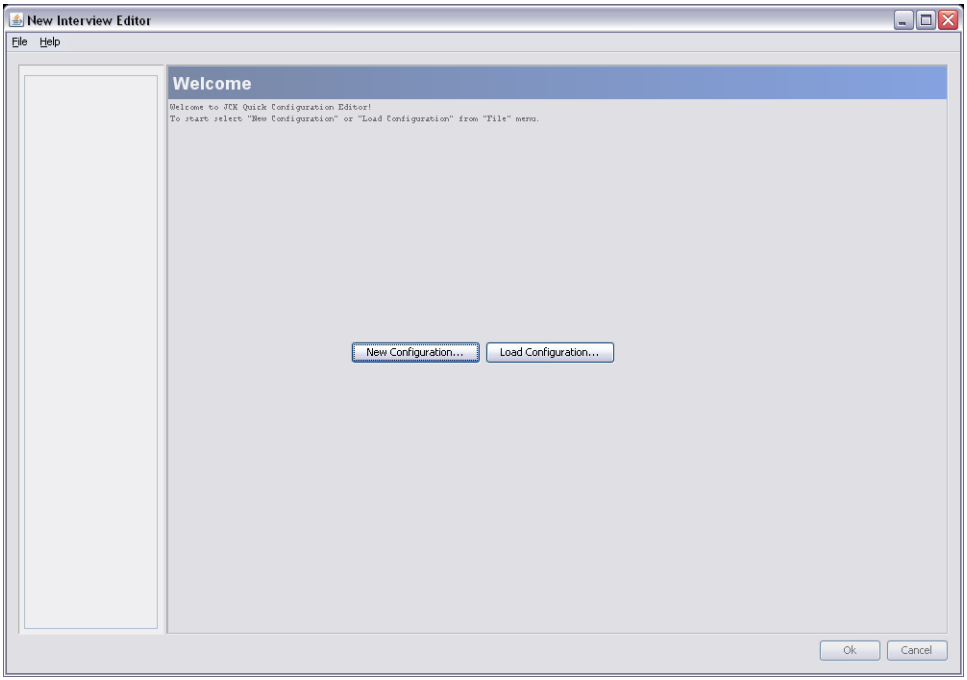
QCE is included with the JCK installation. JCK also installs the following scripts for each platform that can be used to start QCE:

Note – Make sure JAVA_HOME environment variable is set or Java launcher can be found via the PATH environment variable before running QCE start scripts.

- For Solaris environment use `<JCK-HOME>/solaris/bin/jckconfigedit.sh` script
- For Linux environment use `<JCK-HOME>/linux/bin/jckconfigedit.sh` script
- For Windows environment use `<JCK-HOME>\win32\bin\jckconfigedit.bat` batch file

Command line options are not supported in this version of QCE.

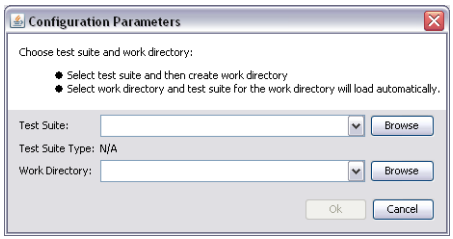
When you run QCE, you will see the initial screen in which you can select to either create a new configuration for a test suite or load the existing configuration for editing.



▼ To create a new configuration

1 **In the initial screen of the QCE, click** `New Configuration`**.**

The `Configuration Parameters` dialog box opens.



2 **In the** `Configuration Parameters` **dialog box, you can perform one of the following:**

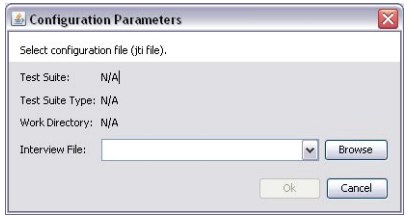
- **Select a test suite and create a work directory**
- **Select a work directory and the test suite will be selected automatically**

Note – For both options you can click Browse if the item is not listed.

▼ To edit an existing configuration

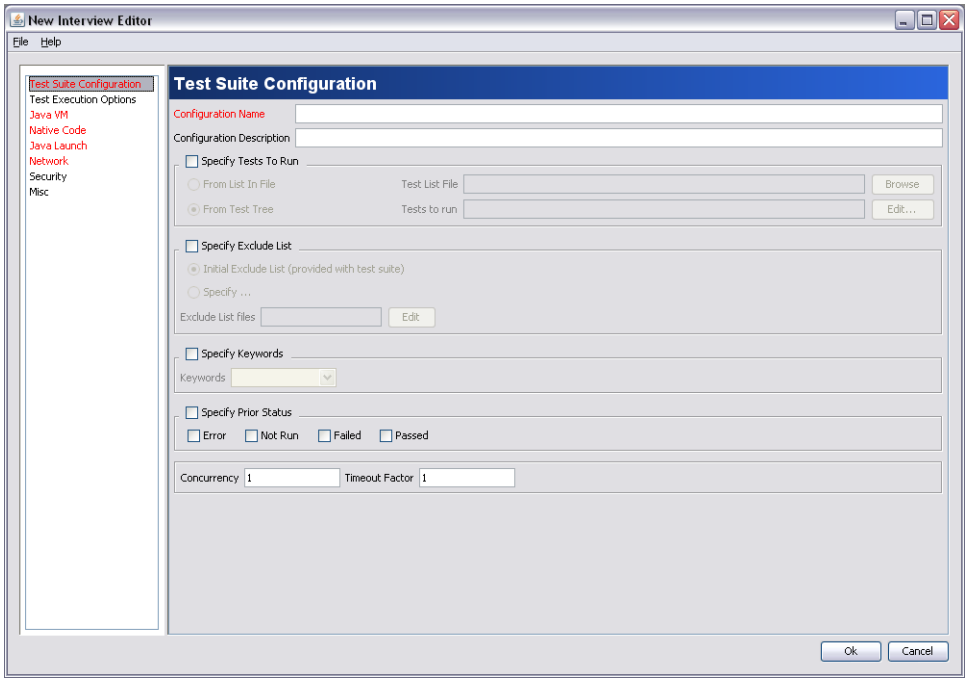
1 **In the initial screen of the QCE, click** `Load Configuration`**.**

The `Configuration Parameters` dialog box opens.



2 **In the** `Configuration Parameters` **dialog box, select the configuration file (JTI) that you want to edit. Click Browse if the file is not listed.**

Once you create a new configuration or load the existing one for editing, the main window of the Quick Configuration Editor opens. The left pane contains a list of option groups. When you select an option group, all options of the selected group are displayed in the right pane.



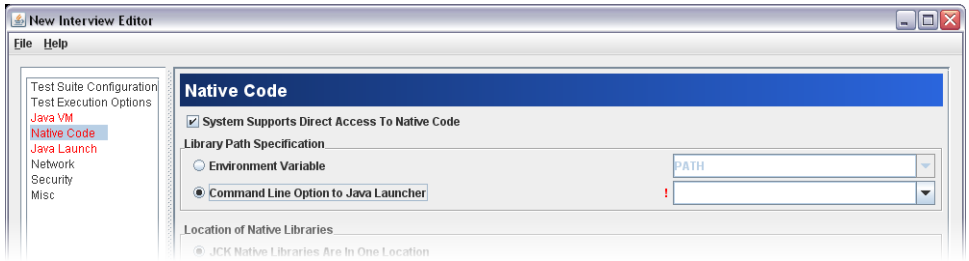
F.1.2 Tips for Using Quick Configuration Editor

This chapter explains some concepts of QCE and provides some tips for working with the editor.

- Fields marked with red color and with an exclamation mark denote mandatory options.

Note – The number of mandatory fields in the interview configuration depends on the selected "tests to run". For example, if you select only *api/javax_swing* tests, the Network options become optional.

- Option groups in the left pane marked with red denote that mandatory options in this group do not have appropriate values, thus making the configuration incomplete.



- You can edit the options in different option groups in any order.
- To save the configuration, click `File` > `Save` or `File` > `Save As`.
- To exit the editor without saving the configuration, click `Cancel`. To exit the editor saving the configuration, click `OK`.

F.2 JCK Test Extractor Tool

The JCK Test Extractor is a command-line tool that can extract sources of a single test from the test suite. You can use this tool for tracing tests, reproducing failures or performing other tasks for which having a set of test sources would simplify the process.

Note – Extracted and compiled tests can not be used for certification process.

To use the Extractor tool, run it with the following command-line options:

```
java –Xmx128m –Xms128m –cp <jck>/lib/jtjck.jar com.sun.jck.lib.extracttest.Main
[options] testURL
```

where options are as follows

- -tckRoot <path>

  <path> is the path to the TCK root to extract the selected test from (default - current)
- -o <path>

  <path> is the path to the output directory (default - current)

Note – Refer to the glossary for a test URL definition.

Note – To simplify the tool usage, run one of the following platform dependent scripts:

- <jck>/solaris/bin/testextractor
- <jck>/linux/bin/testextractor
- <jck>/win32/bin/testextractor

EXAMPLE F–1 Usage Example

To extract the sources for the api/signaturetest/sigtest.basic.html#basic test to the work directory, execute the following command:

```
java -Xmx128m -Xms128m -cp <jck>/lib/jtjck.jar com.sun.jck.lib.extracttest.Main
-tckRoot <jck> -o <workdir> api/signaturetest/sigtest.basic.html#basic
```

or you can use the script corresponding to your platform:

<jck>/{linux|solaris|win32}/bin/testextractor -o <workdir>
api/signaturetest/sigtest.basic.html#basic

Note – In some cases you will not be able to compile extracted sources to binary form without using special tools, which may be privately owned by Sun Microsystems (e.g. files with *.jasm and *.jcod extensions).

Note – For information about Test Extractor tool issues and limitations, refer to the Known Issues section of JCK Release Notes.

Glossary

active agent A type of test agent that initiates a connection to the JavaTest harness. Active test agents you 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 harness agent.

agent monitor The JavaTest harness 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 harness agent.

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.

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

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.

automated tests (AKA 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 might also be considered a set of objects that 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 might be a specialization (a subclass) of another (one of its superclasses), might be composed of other classes, or 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

configuration Information about your computing environment required to execute a Technology Compatibility Kit (TCK) test suite. The JavaTest harness uses a configuration interview to collect and store configuration information.

Configuration Editor The dialog box used by JavaTest harness 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. The JavaTest harness creates a test environment from this information that it uses to execute tests.

configuration value Information about your computing environment required to execute a TCK test or tests. The JavaTest harness uses a configuration interview to collect configuration values.

distributed tests Tests consisting of multiple components that are running on both the device and the JavaTest harness host. Dividing test components between the device and JavaTest harness is often used for tests of communication APIs, tests that are heavily dependent on external resources, tests designed to run on devices with constrained resources such as a small display, and data transfer tests.

environment See test environment.

equivalence class partitioning A test case development technique that 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 to certify compatibility. The JavaTest harness uses exclude list files (`*.jtx`) to exclude from 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 might 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.

Executive Committee (EC) Composed of Members who guide the evolution of the Java technologies, representing a cross-section of both major stakeholders and other Members of the Java Community.

Expert A Member representative (or an individual who has signed the IEPA) who has expert knowledge and is an active practitioner in the technology covered by the JSR.

Expert Group The group of Experts who develop or make significant revisions to a Specification.

Individual Expert Participation Agreement (IEPA)	An agreement between Sun Microsystems and an individual that allows that individual to serve on an Expert Group at the invitation of the Specification Lead. There is no fee associated with the IEPA and it is valid until the Expert Group disbands. The IEPA allows individual technical experts who do not represent companies or organizations to participate on Expert Groups.
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 if the implementation passed or failed the test.
Java Archive (JAR) file	A platform-independent file format that combines many files into one.
Java Community Process (JCP)	An open organization of international Java community software developers and licensees whose charter is to develop and revise Java specification (Specification)s, and their associated reference implementation (RI), and Technology Compatibility Kit (TCK).
Java™ platform. Standard Edition (Java SE™)	A set of specifications that defines the desktop runtime environment required for the deployment of Java technology applications. Java SE platform implementations are available for a variety of platforms, but most notably the Sun Solaristrademark Operating System and Microsoft Windows.
Java Platform Libraries	The class libraries that are defined for each particular version of a Java technology in its Java specification (Specification).
Java specification (Specification)	A written specification for some aspect of the Java technology.
Java Specification Request (JSR)	The actual description of a proposed or final Java specification (Specification) for the Java platform under the Java Community Process (JCP).
Java technology	A Java technology is defined as a Java specification (Specification) and its reference implementation (RI). Examples of Java technologies are the Java SE platform, the Connected Limited Device Configuration (CLDC), and the Mobile Information Device Profile (MIDP).
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 provides flexible and customizable test execution. It includes everything a test architect needs to design and implement tests for implementations of a Java specification (Specification).
JavaTest harness agent	A test agent supplied with the JavaTest harness to run TCK tests on a Java technology implementation where it is not possible or desirable to run the main JavaTest harness. See also test agent, active agent, and passive agent.
JCP	See Java Community Process (JCP).
JCP Specification Page (Spec Page)	Each Specification approved for development or revision will have a dedicated public web page established on the JCP Web Site to contain a history of the passage of the Specification through the JCP, including a record of the decisions, actions, and votes taken by the EC with respect to the draft Specification.
JCP Web Site	The web site where anyone with an Internet connection can stay informed about JCP activities, download draft and final Specifications, and follow the progress of Specifications through the JCP.

Maintenance Lead (ML) The person or organization responsible for maintaining an existing Java specification (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.

passive agent A JavaTest harness component that runs the server portion of distributed tests. It is a type of test agent that must wait for a request from the JavaTest harness before it can run tests. The JavaTest harness initiates connections to passive agents as required. It opens a server socket on the specified port for communications with the JavaTest harness. See also test agent, active agent, and JavaTest harness agent.

prior status A JavaTest harness selection criteria 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 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) service.

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

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

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

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 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 A test that 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 specification (Specification).

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.

TCK coverage file A file used by the Java CTT Spec Trac tool to track the test coverage of a test suite during test development. It binds test cases to their related assertions in the specification. The bindings make it possible to generate statistical reports on test coverage.

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

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

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 specification (Specification)'s compatibility rules. A single test can contain multiple test cases. Accompanying information can include test documentation, auxiliary data files, or other resources used by the source code. Tests correspond to assertions of the specification.

test agent An 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 harness agent.

test cases A small test that is run as part of a set of similar tests. Test cases are implemented using the JavaTest harness *MultiTest* library class. A test case exercises 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 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 tool-style tags.

test environment A collection of configuration values derived from environment entries in the configuration file that provide information used by test suite specific plugin code about how to execute and run each test on a particular platform. When a test in a test suite is run, the JavaTest harness gives the testscript a test environment containing environment entries from configuration data collected by the c onfiguration editor. See configuration.

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 are handed off to the JavaTest harness for further processing.

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

test script A Java technology software class whose job is to interpret the test description values, run the tests, and report the results 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 with the test harness to verify compliance of the licensee's implementation to a Java specification (Specification). Every Technology Compatibility Kit (TCK) contains one or more test suites.

test URL The test URL is tied closely to the test finder and the underlying construction of any particular testsuite. JavaTest will internally convert the path to a test from a platform-specific format into a common internal format - a test URL.

There are two differences between the platform-specific version and the internal format:

- the path is made relative to the root of the test suite
- the path separator is translated to a forward slash

A test URL consists of three components:

- the folder names relative to the test suite root
- the file name which holds the test description for the test
- the optional identifier within the file

  For example: api/javatest/TestSuite.html#getName

  where javatest — folder, TestSuite.html — file, and getName — id.

JavaTest does case insensitive matching when dealing with test URLs. It will preserve case when possible to increase readability for the user. White space is never allowed in the test URL.

The folder names you select must use only these characters:

- English alphabet A-Z (mixed case permitted)
- digits 0-9
- underscore character '_'
- dash character '-'
- period character '.' (deprecated)
- open and close parentheses (deprecated)

Do not use any of the deprecated characters, they are for backwards compatibility and may be unsupported in future.

The filename may contain the same set of characters as the folder names. Note that when the result file (JTR) is created, the text after the last period will be lost. The test identifier can only contain the following characters:

- English alphabet A-Z (mixed case permitted)
- digits 0-9
- underscore character '_'

The URL identifies a location relative to the top of the test suite. A test URL should identify a "real" location in the filesystem and in the file identified (if applicable). If a HTMLTestFinder were to be used, and a test named api/java_lang/Character/index.html#foo was found, the path to the index.html file should be valid and there should be an anchor named "foo" in that file.

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

K

L

M

N

O

P

U

V

W