JavaTest™ Harness 4.5

Command-Line Interface User’s Guide

December 2013

Contents

Preface

This manual describes how to use the command-line interface provided by the JavaTest™ harness (the harness) to run tests of the test suite, browse results, write reports, and audit test results.

The harness provides two User's Guides, the *Graphical User Interface User's Guide* and the *Command-Line Interface User's Guide* . If your test suite uses the JavaTest agent to run tests, the *JavaTest Agent Users' Guide* might also be included.

This User's Guide is also provided by the harness in an online version. The online version of the User's Guide differs from the PDF in the following areas:

- In the online version, all of the harness User's Guides are merged into a single document.
- In the online version, the search function provides a list and ranking of all matching text strings found in the complete harness documentation set.
- The online version can be opened from the command line in a stand alone viewer without opening the GUI.
- In the online version, hypertext links and navigation bars are used instead of page and section references.

Note – Displaying the online version of the User's Guide does not require the installation of any additional software (such as a web browser). The viewer is provided by the harness.

Before You Read This Book

To fully use the information in this document, you must have a thorough knowledge of the topics discussed in the documentation delivered with your test suite.

How This Book Is Organized

Chapter 1 describes the features of the command-line interface provided by the JavaTest harness.

Chapter 2 describes the basic topics that the user should be familiar with before using the command-line interface.

Chapter 3 provides a description of the types of commands and command formats used in the command-line interface.

Chapter 4 describes the commands used to setup and modify a configuration used by the harness.

Chapter 5 describes the commands used to perform tests from the command line.

Chapter 6 describes commands used to specify the properties of the GUI.

Chapter 7 describes the information commands used display online information without starting the harness.

Chapter 8 describes the legacy commands that the harness supports.

Chapter 9 describes the various special utilites provided by the harness.

Chapter 10 provides a basic troubleshooting guide.

Using System Commands

This document does not contain information on basic system 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 environment
- Other software documentation that you received with your system

Typographic Conventions

This User's Guide uses the following typographic conventions:

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`

`AaBbCc123`	What you type, when contrasted with on-screen computer output	`%` **`su`**
		`Password:`

AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this.
	Command-line variable; replace with a real name or value	To delete a file, type `rm` *filename*.

Shell Prompts

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

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

Related Documentation

The following documentation provides additional detailed information about the JavaTest harness:

Application	Title
JavaTest harness GUI	*Graphical User Interface User's Guide;*
JavaTest harness agent (optional)	*JavaTest Agent User's Guide*

Accessing Documentation Online

The Oracle Technology Network enables you to access Java ME technical documentation on the Web:

`http://download.oracle.com/javame/index.html`

We Welcome Your Comments

We are interested in improving our documentation and we welcome your comments and suggestions. Provide feedback at `javamedocs_us@oracle.com`.

1

Command-Line Interface

The test harness provides two interfaces, a Graphical User Interface (GUI) and a command-line interface.The command-line interface provides test harness functionality for configuring and executing tests as well as creating reports without using the GUI. This enables you to use the harness to run tests in build scripts and other automated processes. See the *Graphical User Interface User's Guide* for a description of the harness GUI.

Features

Features of the command-line interface include the following capabilities:

- **Enables configurable testing** - Users can run tests from the command line on a variety of test platforms (such as servers, workstations, browsers, and small devices) with a variety of test execution models (such as API compatibility tests, language tests, compiler tests, and regression tests).
- **Runs tests on small systems** - Users can use an agent (a separate program that works in conjunction with the harness) to run tests on systems that can't run the harness.
- **Generates HTML reports** - Users can generate HTML reports for the test run.
- **Provides online User's Guide** - Users can launch a viewer that displays the online version of the User's Guide. The User's Guide describes how to use the harness to run test suites and evaluate test results. The viewer provides full-text search and keyword search capabilities.

Configuration Information

The GUI Configuration Editor collects the required configuration information about how tests are run on a specific test platform and saves that information in a configuration file (`.jti` file). By using configuration files, the harness can run programs on a variety of Java platforms. The harness writes the test results in the work directory associated with that configuration and test suite. See the Glossary for detailed descriptions of the terms `.jti` file, *work directory*, and *configuration file*. See the *Graphical User Interface User's Guide* for a description of the GUI Configuration Editor.

Legacy Environment Files and Parameter Files

For backwards compatibility, older test suites can continue to use environment and parameter files in the command line. See the Glossary for detailed descriptions of the terms `.jte` file and `.jtp` file.

See the *Graphical User Interface User's Guide* for a description of using a legacy configuration file to create current configuration file.

Online Documentation

The harness provides extensive online documentation that is available from the command line. To display the available command-line options, type the following at a system prompt:

```
java -jar testsuite/lib/javatest.jar -help
```

See Information Commands for detailed information about using special `-help` options to search for and display command-line information.

To display the online User's Guide without starting the GUI, type the following at a system prompt:

```
java -jar testsuite/lib/javatest.jar -onlinehelp
```

Note – Include the path of the directory where the `javatest.jar` file is installed (represented as *`testsuite`*/lib in the example). The `javatest.jar` file is usually installed in the test suite `lib` directory when the harness is bundled with a test suite.

Before Starting the Harness

Before you start the harness on a test system, you must have a valid test suite and Java Development Kit 6.0 or higher installed on your test system. See your test suite documentation for information about installing the test suite on your test system. Refer to `http://www.oracle.com/products` for information about installing the current Java Development Kit on your test system.

You must also understand how your test group uses or intends to use the harness in its test system. For example, consider the following questions:

- Does your test group use the harness and one or more agents to run distributed tests?

  If you use an agent to run tests, you must also install the agent on the platform being tested. If you intend to use the JavaTest agent, see *JavaTest Agent User's Guide* for detailed information about installing the harness agent on a test platform. If you use a custom agent, refer to your test suite documentation for detailed information about installing the agent.
- Does your test group use configuration files and templates from a central location, or does it use individual configuration files customized for each test run?

  If your group intends to use configuration files and templates from a central location, the location must be configured for the harness to access the template files.
- Does your test group run the harness from a central location or from local installations in the test system?

If your group uses a central location for the test system, you must be able to access the test system at that location.

Installation and Runtime Security Guidelines

It is extremely important to note that the JavaTest installation and runtime system is fundamentally a developer system that is not specifically designed to guard against any malicious attacks from outside intruders. If sample code or tests you author make a network call, you can expose the JavaTest operating environment during execution. For this reason, it is critically important to observe the precautions outlined in the following security guidelines when installing and running JavaTest.

The harness itself is self-contained in `javatest.jar`. If desired, the following optional parts of the binary distribution can be deleted:

- The directories `linux`, `solaris`, and `win32` each contain a subdirectory named `bin`, which contains a script named `javatest`. All of these directories and files are provided for convenience and can be deleted without harm. (If the `javatest` launch script is deleted you can start the harness by pointing a JVM at the `javatest.jar` file.)
- Examples in the `examples/` directory, which is an optional binary component and may not be present.
- Documentation in the `doc/` directory.
- `lib/jt-junit.jar` which is an optional binary component and may not be present.

To maintain optimum network security, JavaTest can be installed and run in a "closed" network operating environment, meaning JavaTest is not connected directly to the Internet, or to a company Intranet environment that could introduce unwanted exposure to malicious intrusion. This is the ideal secure operating environment when it is possible.

JavaTest does not require an "Intranet" connection that supports network connections to systems outside the JavaTest architecture to intra-company resources, but, for example, some Java ME applications in a test suite might use an HTTP connection. If JavaTest or applications launched from JavaTest are open to any network access you must observe the following precautions to protect valuable resources from malicious intrusion:

Install JavaTest behind a secure firewall that strictly limits unauthorized network access to the Java Runtime file system and services. Limit access privileges to those that are required for JavaTest usage while allowing all the I-directional local network communications that are necessary for JavaTest functionality. The firewall configuration must support these requirements to run JavaTest while also addressing them from a security standpoint.

Follow the principle of "least privilege" by assigning the minimum set of system access permissions required for installation and execution of JavaTest.

Do not store any data sensitive information on the same file system that is hosting JavaTest.

To maintain the maximum level of security, make sure the operating system patches are up-to-date on the JavaTest host machine.

2

Important Security Information

It is important to note that the harness is fundamentally a developer system that is not specifically designed to guard against malicious attacks. This section describes known harness vulnerabilities that can be exploited by authorized or unauthorized users or others who have access to the network where the harness runs. When installing and operating the harness, consider these vulnerabilities and take action to mitigate potential threats.

Note: See the *Release Notes* for additional security information.

File Access Risks

The harness reads, writes, and deletes files and can therefore potentially expose or damage information stored in files.

Path Vulnerabilities

The harness and its utilities do not prevent users from accessing sensitive files that are accessible from the computer on which the harness is running - including attached/mounted remote filesystems. A harness user can therefore directly or indirectly examine file contents, overwrite contents, or delete them. To mitigate against accidental or intentional misuse of sensitive files, consider setting file permissions to give the least feasible access to harness users.

There are no direct options in the harness for removing a file, directory, or folder, but the harness can automatically remove files that it believes it owns and are corrupt or in need of replacement. Automatic removal or replacement generally occurs only inside the harness preferences metadata storage area (`~/.javatest` or `user_home\.javatest`) and inside work directory locations directly designated by the user.

There are at least two ways a user can indirectly give the harness access to an arbitrary file that the harness might automatically remove or replace.

- A configuration (`.jti`) file can contain a "hint" that enables the harness GUI or CLI to find what the harness *assumes* is a work directory.
- A desktop file in the preferences metadata storage area can contain a path to what the harness *assumes* is a work directory. The harness GUI or CLI automatically opens the desktop file.

File Creation Permissions

The harness is intended to be run by users who are trusted to properly handle files they have access to. The harness does not attempt to override any file permissions that are in place to protect files or restrict the user. Additionally, the harness creates files with whatever permissions are the default for the user running the harness (on Unix systems, the `umask`, etc). If you want to restrict access to files created by the harness, ensure that user default file creation permissions are set accordingly.

Network Access Risks

The harness and related components are intended to be run in a semi-trusted environment. Never expose the harness or its utilities directly to the Internet, which provides a path for malicious intrusion. The most secure environment is a standalone machine with no network access, if this is feasible for the testing you are performing. However, because the test harness agent configuration and some test suites require a network, a more realistic environment is a closed intranet, ideally one that is physically isolated from organizational intranets and sensitive information.

Remote Agent Risks

The test harness includes the JavaTest agent, which is a remote execution framework. The agent requires open communication ports on the JavaTest host computer and on the remote computer (device); therefore you must ensure that both machines are protected from malicious attack. For the most secure operation, connect the host and remote computers only to a protected intranet on a physically isolated network.

Host Name (DNS) Lookup Vulnerabilities

The harness sometimes uses Domain Name Services (DNS), or similar from your system, to transform a host name into an IP address. Invalid information returned by this lookup might cause the harness to connect to an address you did not intend. When possible, the harness GUI displays the returned IP address. Users should verify that the address is appropriate, for example, lies within a range the corresponds to your network. If you need assistance validating IP addresses, contact your network administrator.

Command Injection Vulnerability

As part of the normal process of harness configuration and testing, there are places where the user is allowed to specify arbitrary commands (with parameters) to execute - for example, to start a test, to start a Java virtual machine being tested, or to start a service. The harness does not screen these commands for malicious or accidental usage. The harness offers no protection from the effects of any command initiated by a user or a test. Be sure that users are trained and trusted to specify proper commands properly, and to verify that commands issued by test suites are harmless. An incorrect or malicious command can cause serious damage.

The harness does not attempt to run commands with higher privileges than that of the current user or the privileges inherited when the new command/process executes.

3

Command-Line Summary

You can use commands in the command line or as a part of a product build process to configure the harness, run tests, write test reports, and start the GUI using specific configuration values.

The harness executes the commands from left to right in the sequence that they appear in the command string. Include commands in the command string as though you were writing a script. The harness does not restrict either the number of commands or the groups of commands that you can use in a command string.

> javatest [*Setup commands*] [*Task commands*] [*Desktop commands*] [*Information commands*]

The commands are included as a formatted set in the following sequence:

1. Setup commands - Required by task commands to set values used for the test run and to set specific values used when performing other tasks. Setup commands must precede the task or desktop commands. Setup commands can be used to set specific values (without a task command) when starting the GUI.

2. Task commands - Required to run tests, and write reports. Task commands require one or more preceding setup commands.

3. Desktop commands - Use in place of the task commands to start the GUI with a new desktop or to specify status colors used in the GUI. Setup commands are optional when using Desktop commands.

4. Information Commands - Use information commands to display command-line help, online help, or version information without starting the harness. Information commands do not require any other commands on the command line.

For additional information about using the command-line interface, see the following topics:

- About Command-Line Examples
- Formatting a Command

- Using Command Files
- Index of Available Commands

About the Command-Line Examples

This section provides many examples of command-line operations in the following basic sequence:

> > *javatest* [*Setup commands*] [*Task commands*]

In the examples, the following presentations are used:

- > represents the command prompt. For Unix systems, the command prompt can be either a shell prompt, such as %, or a user defined value. For win32 systems, the command prompt can be **c:** or another appropriate drive identifier.
- *javatest* represents the command or commands that your test suite would use to start the harness.

See Setup Commands for commands and examples used to set up or change specific values in a configuration.

See Task Commands for commands and examples used to perform tasks from the command line.

Formatting a Command

You can use any one of the following formats to include commands on the command line:

- Command Options Format
- Single String Arguments Format
- Command File Format

All formats are used to accomplish the same tasks. Use the format that you prefer or that is easier to use. See Index of Available Commands for a complete listing of available commands.

Command Options Format

In the command options format, commands are preceded by a dash (-), act as options, and do not use command terminators. Enclose complex command arguments in quotes. Use this format when long lists of commands are included in a command line.

Example:

```
> javatest -open default.jti -runtests
```

Single String Arguments Format

If you are setting several command options, you might want to use the single string arguments format. In the single string arguments format, one or more commands and their arguments can be enclosed in quotes as a single string argument. Multiple commands and arguments in the string are separated by semicolons.

Example:

```
> javatest "open default.jti; runtests"
```

Command File Format

If you are setting a series of commands and options, you can use the command file format. Using a command file enables you to easily reuse the same configuration.

In the command file format, a file containing a series of commands and their arguments is included in the command line by preceding the file name with the at symbol (@).

Example:

```
> javatest @mycmd.jtb -runtests
```

Refer to Using Command Files for detailed information about using and creating command files.

Using Command Files

A command file is a text file that contains one or more commands used by the harness from the command line or as a part of a product build process. You can place combinations of configuration settings and commands in the command file and use it to repeatedly perform the following actions:

- Perform test runs
- Write test reports

The advantage of using the command file format is that it is easy to use a complex, persistent, repeatable set of commands in a command line.

The commands used in a command file are a formatted set of commands, executed in the sequence that they appear in the command string. Use the commands in the command file as you would if you were writing a script. See Formatting a Command for a description of the formats you can use.

The following topics provide additional detailed information:

- Creating a Command File
- Examples of Using Command Files

Creating a Command File

Use the single string arguments format style to write commands in a text file. See Formatting a Command for detailed information.

Command files can contain blank lines and comments as well as lines with commands and their arguments. The following table describes the contents of a command file.

TABLE 1 Command File Contents

File Contents	Description
Comments	Comments can begin anywhere on a line, are started by the pound symbol (#), and stop at the end of the line. Example: `#File contains commands`
Commands	Commands are executed in the sequence that they appear in the file (for example, setup commands must precede task commands). Commands used in the file must be separated by a semicolon (;) or a new line symbol (#). The # symbol acts as a new line character and can terminate a command. Examples: `open` *default*`.jti; #opens file` `-set host` *mymachine*
Command Arguments	Arguments that contain white space must be placed inside quotes. Use a backslash (\) to escape special characters such as quotes (" ") and backslashes (\).

After writing the commands, use a descriptive name and the extension `.jtb` to save the text file. Choose a file name that helps you identify the function of each command file.

Examples of Using Command Files

In the following examples, a command file (*mycommandfile*`.jtb`) is used to override the `localHostName`value and the tests specified in the existing configuration.

The following three examples are provided:

- Example Command File Contents
- Command Line Using the Example Command File
- Changing Values After the Example Command File is Set

Note – If you attempt to run these examples, you must replace *mytestsuite*`.ts`, *myworkdir*`.wd`, and *myconfig*`.jti` with test suite, work directory, and `.jti` names that exist on your system. You must also modify the contents of the example command file for your configuration file and test suite. Win32 users must change / file separators to \ to run these examples.

Example Command File Contents

The following lines are the contents of the example command file, *mycommandfile*`.jtb`:

```
#File sets localHostName and tests
 set jck.env.runtime.net.localHostName mymachine;
 tests api/javax_swing api/java_awt
```

Note – The `-set` and `-tests` command forms are not used in the command file. Command files only use the "Single String Arguments Format."

See Setting Specific Configuration Values for additional examples of using the `set`command. See Specifying Tests to Run for additional examples of using the `tests` command.

Note – See About the Command-Line Examples for a description of the use of > *javatest* in the following example. See Command-Line Overview for a description of the command line structure. See Formatting a Command for descriptions and examples of the following command formats.

Command Line Using the Example Command File

In the following examples, a test suite (*mytestsuite*`.ts`), work directory (*myworkdir*`.wd`), and configuration file (*myconfig*`.jti`) are opened, and the command file (*mycommandfile*`.jtb`) is read and executed before running tests.

Command Options Format Example

```
> javatest -open myconfig.jti @mycommandfile.jtb -runtests
```

Single String Arguments Format Example

```
> javatest "open myconfig.jti; @mycommandfile.jtb; runtests"
```

Changing Values After the Example Command File is Set

You can also change values after the command file is set:

Command Options Format Example

```
> javatest -open myconfig.jti @mycommandfile.jtb -excludeList
myexcludelist.jtx -runtests
```

Single String Arguments Format Example

```
> javatest "open myconfig.jti; @mycommandfile.jtb; excludeList
myexcludelist.jtx; runtests"
```

Formatting Configuration Values for `editJTI` or `-set`

The following table identifies and describes the types of questions supported by the harness configuration interview. as well as provides a description of the format required to set the value in the command line.

TABLE 2 Types and Values of jti Questions

Question Type	**Description**
Choice Question	This question is used to get a selection from a finite list of possible values. For example, in the question "Which protocol would you use," where the only possible responses are TCP or UDP. These questions are usually displayed in the Configuration Editor and the Template Editor as radio buttons where you can only select one button at a time. The following is an example of the format used to set this configuration value in a command line: `set` *My-testsuite*`.cipher 3DES` The value supplied is case sensitive. This type of question appears in the Configuration Editor as a set of radio buttons or single-selection list of choices.

File Question	This question is used to represent a file path. It may be absolute or relative, depending on the context of the question. See the question's More Info for information about the requirements. The value used is generally a platform specific path. The question may or may not check to see that the value is valid before it is accepted.
	The following are examples of the format used to set this configuration value in a command line:
	`set` *My-testsuite.myfile* `c:\`*foo*`\`*bar*`.txt`
	`set` *My-testsuite.myfile /tmp/bar*`.txt`

File List Question	If none of the file names have embedded spaces, you can use a space-separated list of file names. If any of the file names in the list have embedded spaces, use a newline character to terminate or separate all of the filenames.
Floating Point Question	This question is primarily used to enter fractional numbers, but can also accept whole numbers. It might be used to collect values such as a timeout factor in seconds, where a value similar to 1.5 might be entered. It usually appears as a type-in field in both the Configuration Editor and the Template Editor. The question might be set to reject values outside a specified range. See the More Info in the Configuration editor or the Template Editor for guidelines regarding the values required for a specific question. The following is an example of the format used to set this configuration value in a command line: `set` *My-testsuite*`.delay 5.0` The value is evaluated using the current locale (for example, in European locales, enter 5,0).

TABLE 2 Types and Values of jti Questions

Integer Question	These questions are commonly used to get port numbers or to specify the number of times to do an action. The answers are always restricted to whole numbers and might have further restrictions that prevent you from using certain ranges of numbers (such as negative numbers). You might also be restricted to using only a particular set of pre-determined numbers. In the Configuration Editor and in the Template Editor, these usually appear as plain type-in fields or may be a field which has up and down (spinner) controls to select the number. The following is an example of the format used to set this configuration value in a command line: `set` *My-testsuite*`.port 5000` Localized values can be used. For example, 5,000 is acceptable in a US locale.
IPAddress Question	The standard textual representation of the IP Address, as defined by Internet Engineering Task Force (IETF). A typical IPv4 address string would be "192.168.1.1".
Multi Choice Question	This question is used when a selection of choices from a finite set of possible values is required from the user. In the Configuration Editor and Template Editor, this question type resembles the Choice question in that it has a list of choices with checkboxes. The difference between them is that in a Multi Choice question, you can select more than one checkbox. The values that you use in the `set` command must be separated by whitespace (newline, space or tab) and must be the absolute new settings for the question. The values that you use are absolute settings and cannot be based on the default or previous value. You must enter the full list of values that you want to turn on (corresponding to the items checked in the Configuration Editor or Template Editor representation).
Property Question	This question enables you to change a grouped set of property values for a test suite. In the Configuration Editor and Template Editor, this question type enables the user to view multiple property settings and to change read-write values. The user can also copy both key and values to the clipboard. Value types supported by this question include integer, float, yes-no, file, and string. In the Configuration Editor and Template Editor, the question can provide suggestions to users for the integer, float, file, and string value types that it contains. The following are examples of the format used to set this configuration value in a command line: `set` *question-key property-name*:*new-value* `set` *question-key=property-name*:*new-value* Each `set` statement for a Property Question value must contain the question key, the property name, and a new value. The use of an equals sign or a space is determined by the tool and the type of value that you are changing. You cannot change multiple values within a single set statement.

TABLE 2 Types and Values of jti Questions

String Question	This question contains a generic field that enables you to enter a value. While this is much less restrictive than the integer or the file question, there might be some restrictions on the values that you can enter. In the most restrictive cases, you can only use values that are predefined. The More Info displayed in the Configuration Editor and the Template Editor might indicate what constitutes a legal value in this field. View the More Info for detailed information about the values that you can enter in this field. The following is an example of the format used to set this configuration value in a command line: `set` *My-testsuite*`.url http://`*machine*`/`*item*
String List Question	This question is used when multiple discrete string values are required from the user. It is usually shown as a interface which allows you to enter a string then add it to a list. The list of strings provided in the `set` command become the new absolute answer to the question, not appended values. See Using Newlines Inside Strings.

YesNo Question	This question is used when either a positive or a negative response is needed from the user.
	The following are examples of the format used to set this configuration value in a command line:
	`set` *My-testsuite*`.needStatus Yes`
	`set` *My-testsuite*`.needStatus No`
	The values are case sensitive and a lower-case value of yes or no is not acceptable.

Using Newlines Inside Strings

When setting values of configuration questions in the command line, the internal command parser accepts newlines inside strings if they are preceded by a backslash.

Depending on the shell you use, it might or might not be possible to enter this directly on the command line.

If you want to set values with embedded newlines, create a harness batch command file, and put the set commands (and any other commands) in that file. In the batch file, you can enter strings with embedded escaped newlines, as in the following example:

```
# switch on verbose mode for commands
verbose:commands
# open a jti file
open /home/user1/tmp/idemo.11mar04.jti
# set a list of files
set demo.file.simpleFileList /tmp/aaa\
/tmp/bbb\
/tmp/ccc
# set a list of strings
```

```
set demo.stringList 111\
2222\
3333
```

On Solaris, using the Korn shell, you can simply put newline characters into strings.

Example:

```
$JAVA \
-jar image/lib/javatest.jar \
-verbose:commands \
-open /home/user1/tmp/idemo.11mar04.jti \
-set demo.file.simpleFileList /tmp/aaa
/tmp/bbb
/tmp/ccc
```

Extended Command-Line Examples

This section provides extended examples of command-line operations.

To use the following examples on your system, you must use classpaths and directory names appropriate for your system.

Example 1

```
java -jar lib/javatest.jar -verbose -testSuite /tmp/myts \
-workdir -create /tmp/myworkdir -config /tmp/my.jti \
-runtests -writereport /tmp/report
```

This combination of commands does the following, in this order:

1. Tells the harness to be verbose during test execution.
2. Opens the test suite `/tmp/`*myts*.
3. Creates a work directory named `/tmp/`*myworkdir*.
4. Uses *my*`.jti` as the configuration settings.
5. Executes the tests (as specified by the configuration).
6. Writes a report to `/tmp/report/` after test execution.

Example 2

```
java -jar lib/javatest.jar -startHttp -testsuite /tmp/myts \
-workdirectory /tmp/myworkdir -config /tmp/my.jti \
-runtests -writereport /tmp/report -set foo.bar 4096 \
-runtests -writereport /tmp/report1
```

This combination of commands does the following, in this order:

1. Tells the harness to start the internal HTTP server.
2. Opens the test suite `/tmp/`*myts*.
3. Uses a work directory named `/tmp/`*myworkdir*.
4. Uses *my*`.jti` as the configuration settings.
5. Executes the tests (as specified by the configuration).
6. Writes a report to `/tmp/report/` after test execution.
7. Changes a configuration value (not written to JTI file).
8. Runs tests again.
9. Writes a new report in `/tmp/`*report1*.

Example 3

```
java -cp lib/javatest.jar:lib/comm.jar \
com.sun.javatest.tool.Main \
-Especial.value=lib/special.txt \
-agentPoolPort 1944 -startAgentPool "testsuite /tmp/myts ; \
workdir /tmp/myworkdir ; config myconfig.jti ; runtests"
```

This combination mixes two styles of command line arguments (quoted and dash-style). It invokes the harness by class name, rather than executing the Java Archive (JAR) file (`-jar`). An extra item is added to the VM's classpath. The following commands are given to the harness:

1. Sets a particular value in the testing environment.
2. Specifies the agent pool port and starts the agent pool.
3. Loads the test suite `/tmp/`*myts*.
4. Opens the work directory `/tmp/`*myworkdir*.
5. Uses the configuration in *myconfig*`.jti`.

6. Runs the tests.

Example 4

```
java -jar lib/javatest.jar -config foo.jti -runtests
```

This command example relies on information in the JTI file to perform the run. Specifically, it tries to use the work directory and test suite locations specified in the JTI file. If either of those are invalid or missing, the harness reports an error. Otherwise, if the configuration in the JTI is complete, the tests are run.

Example 5

```
java -jar lib/javatest.jar -config foo.jti -verbose \
-set test.val1 2007 -runtests
```

This is the same as Example 4 with the exception that it turns on verbose mode and changes the answer of one of the questions in the configuration.

Example 6

```
java -jar lib/javatest.jar -config foo.jti \
-priorStatus fail,error -timeoutFactor 0.1 \
-set test.needColor Yes \
-set test.color1 orange -tests api/java_util -runtests
```

This example extends Example 4 by setting various Standard Values and the answer to particular configuration questions.

Example 7

```
java -jar lib/javatest.jar -testsuite /tmp/foo.jti myts \
-workdirectory /tmp/mywd -config /tmp/myconfig.jti
```

This example stars the GUI. This combination of commands does the following, in this order:

1. Opens the specified test suite.
2. Opens the work directory given (assuming it is a work directory).

3. Opens the configuration file given.
4. Starts the GUI since no execution action is given.

List of Available Commands

The following table describes all of the commands available in command mode.

TABLE 3 Summary of Available Commands

Command	Description
`concurrency`	Specifies the number of tests that are run concurrently.
`env`	Specifies a test environment in an environment file.
`envFile` or `envFiles`	Specifies an environment file (`.jte`) containing test environments.
`excludeList`	Specifies an exclude list file.
`keywords`	Restricts the set of tests to be run based on keywords.
`kfl`	Specifies one or more known failure list (`.kfl`) files.
`open`	Opens a test suite, work directory, or a configuration `.jti` file.
`params`	This command is deprecated.
`priorStatus`	Selects the tests included in a test run based on their outcome on a prior test run.
`set`	Overrides a specified value in a configuration (`.jti`) file.
`tests`	Creates a list of test directories and/or tests to run.
`testSuite`	Specifies the test suite.
`timeoutFactor`	Increases the amount of time the harness waits for a test to complete
`workDir` or `workDirectory`	Opens an exiting work directory, creates a new work directory, or replaces an existing work directory with a new work directory.
`runTests`	Runs tests in command mode.
`writeReport`	Writes test reports in command mode.

4

Setup Commands

Before you can perform a task from the command line, you must first use setup commands to specify a configuration.

After setting up a configuration, you can then modify the values in the configuration for your specific requirements. These changed values override but do not change values in the configuration file. You can use configuration templates from a central resource to run tests on different test platforms and configurations.

Note – See About the Command-Line Examples for a description of use of > *javatest* in the following example.

The setup commands are used in the following sequence in the command line:

> *> javatest* [*initial-setup-commands*] [*set-specific-values*] [*additional-setup-commands*] [*task-commands*]

Setting specific values and additional setup commands are optional.

The task command at the end of the example is also optional. If a task command is not included, the harness uses the specified configuration and any changes set on the command line to open the GUI.

For additional information about using setup commands see the following topics:

- Initial Setup Commands
- Setting Specific Values
- Additional Setup Commands

Initial Setup Commands

Before you can perform tasks from the command line, you must first set up a configuration for the harness to use. You can set up a configuration by performing *at least one* of the following:

1. Specify an existing configuration (`.jti`) file. You are not required to specify either a test suite or a work directory.
2. Specify an existing work directory and a configuration file. You are not required to specify a test suite.
3. Open a test suite, create an empty work directory, and specify a configuration file.

After setting up a configuration, you can then change specific values for your specific requirements. See Setting Specific Values for the commands used to modify the values in the configuration.

You can include commands in any combination on the command line provided the initial set-up commands are specified before any other commands in the command line.

You can use any of the following commands to set up a configuration for the harness to use when performing tasks:

- `config` - Specifies an existing configuration file. See Specifying a Configuration File With `config` for a detailed description of this command.
- `workDirectory` or `workDir` - Specifies an existing work directory or to create a new work directory. See Specifying a Work Directory With `workDir` for a detailed description of this command.
- `testSuite` - Used to specify a test suite. See Specifying a Test Suite With `testSuite` for a detailed description of this command.
- `open` - Specifies a test suite, work directory, configuration file, or parameter file. See Specifying a Test Suite, Work Directory or Configuration With `open` for a detailed description of this command.

Specifying a Configuration File With `config`

To specify the configuration file the harness uses to run tests, use the `config` command.

> *javatest* ... `-config` *path/filename* ... [*task-command*] ...

See About the Command-Line Examples for a description of > *javatest* in the example.

The configuration file might contain default values for the test suite (which contains the tests to be run) and the work directory (where the results are placed).

Test suite and work directory values in the configuration file can be overridden with the `testsuite` and `workdirectory` commands. If the configuration file is a template and does not contain default values for the test suite and work directory, those values must be specified explicitly with the testsuite and workdirectory commands.

See Command-Line Overview for a description of the command line structure.

Detailed Example of `config` Command

In the following example, *myconfig*`.jti` represents a configuration file name that might exist on your system.

Command Options Format Example:

```
> javatest -config myconfig.jti -runtests
```

See Formatting a Command for descriptions and examples of other command formats that you can use.

Specifying a Work Directory With `workdir` or `workdirectory`

Each work directory is associated with a test suite and stores its test result files in a cache. You can use the work directory command to:

- Use an Existing Work Directory
- Create a New Work Directory
- Replace an Existing Work Directory

See Shortcuts to Initialize a Configuration for information about specifying a work directory in the command line.

Use an Existing Work Directory

To use an existing work directory for the test run, include either the `workdir` or `workdirectory` command in the command line:

> *> javatest* ... `-workdir` *path/filename* ... [*task-command*] ...

See About the Command-Line Examples for a description of the use of > *javatest* in the example.

See Command-Line Overview for a description of the command line structure.

See Formatting a Command for descriptions and examples of the following command formats.

Create a New Work Directory

To create a new work directory for the test run, use the `-create` command option:

> *> javatest* ... `-workdir -create` *path/filename* [*configuration-command*] ... [*task-command*] ...

See About the Command-Line Examples for a description of use of > *javatest* in the example. See Command-Line Overview for a description of the command line structure.

The new work directory must not previously exist. You can also use an existing work directory as a template to create a new work directory for the test run. To use an existing work directory as a template, put the template in the command line before the `create` command.

When creating the command string, include the commands in the following sequence:

1. (Optional) Include the commands required to specify the test suite.

See Specifying a Test Suite With `testSuite` for detailed information about the command.

2. (Optional) Include the `-workdir` *path/filename* command required to specify an existing work directory.

3. Include the `workdir` or `workdirectory -create` *path/filename* command.

4. Include the commands required to specify a configuration file.

See Specifying a Configuration File With `config` for detailed information about the command.

5. (Optional) Include the commands required to set specific values.

See Setting Specific Values for detailed information about the available commands.

1. (Optional) Include the `runtests` command.

The results of the test run are written to the new work directory. See Running Tests With `runtests` for detailed information about the command.

Detailed Example of Creating a New Work Directory

In the following example, *myworkdir*`.wd` and *myconfig*`.jti` represent file names that might exist on your system.

Command Options Format Example:

> *javatest* `-workdir` *myworkdir*`.wd -create` *testrun*`.wd -config` *myconfig*`.jti -runtests`

When the tests are run, the harness uses the work directory (*testrun*`.wd`) created by the command line, even if the configuration file (*myconfig*`.jti`) was created using another work directory.

See Formatting a Command for descriptions and examples of other command formats that you can use.

Replace an Existing Work Directory

When you replace an existing work directory with a new work directory, the harness performs the following tasks:

- Deletes the existing work directory and its contents.
- Creates the new work directory using the same name (if the old directory was successfully deleted).

To replace an existing work directory with a new work directory, use the `-overwrite` command option.

> *javatest* ... `-workdir -overwrite` *path/filename* ... [*task-command*] ...

or

> *javatest* ... `-workdir -create -overwrite` *path/filename* ... [*task-command*] ...

The `-create` command option is optional when the `-overwrite` command is used.

See About the Command-Line Examples for a description of use of > *javatest* in the examples.

See Command-Line Overview for a description of the command-line structure.

Detailed Example of Replacing an Existing Work Directory

In the following example, *myconfig*`.jti` represents a configuration file name that might exist on your system.

Command Options Format Example:

```
> javatest -workdir -overwrite testrun.wd -config myconfig.jti -
runtests
```

The harness uses the work directory *testrun*`.wd` created by the command line when the tests are run, even if *myconfig*`.jti` was created using another work directory.

See Formatting a Command for descriptions and examples of other command formats that you can use.

Specifying a Test Suite With `testsuite`

To specify the test suite, use the `testsuite` command:

> *javatest* ... `-testsuite` *path/filename* [*work-directory-command*] [*configuration-command*] ... [*task-command*] ...

See About the Command-Line Examples for a description of the use of > *javatest*.

See Command-Line Overview for a description of the command line structure.

When you want to specify a test suite, include the commands in the following sequence:

1. Include the command required to specify the test suite (`testsuite` *path/filename*).

2. Include the commands required to set up a configuration.

See Set-up Commands for detailed description of the available commands.

3. (Optional) Include a task command such as the `runtests` command.

See Task Commands for a description of the available commands.

Detailed Example of `testsuite` Command

In the following example, *mytestsuite*, *myworkdir*`.wd`, and *myconfig*`.jti` represent file names that might exist on your system.

Command Options Format Example:

> *javatest* `-testsuite` *mytestsuite* `-workdir` *myworkdir*`.wd -config` *myconfig*`.jti -runtests`

See Formatting a Command for descriptions and examples of other command formats that you can use.

Specifying a Test Suite, Work Directory or Configuration (`open`)

To specify a test suite, work directory, or a configuration `.jti` file, use the `open` command:

... `open` *path/filename* ...

Note – See About the Command-Line Examples for a description of the use of > *javatest* in the following example. See Command-Line Overview for a description of the command line structure. See Formatting a Command for descriptions and examples of the following command formats.

Command Options Example:

> *javatest* ... `-open` *path/filename* ... [*task command*] ...

Single String Arguments Example:

> *javatest* ... `;` `open` *path/filename* ; ... [*task command*] ...

Command File Example:

> *javatest* `@`*mycmd*`.jtb` ... [*task command*] ...

In addition to any other commands, for this example the *mycmd*`.jtb` command file must contain the command:
"`open` *path/filename;*"

Refer to Using Command Files for detailed information about creating and using command files.

Setting Specific Values

After you set a configuration (see Initial Set-up Commands), you can specify individual values for a test run that override those in the configuration file.

Note – Values that you specify in the command string override but do not change the values specified in the configuration file.

You can use the following commands to specify individual values for a test run:

- `set` - Sets any value in a configuration file. See Setting Test Suite Specific Values With `set` for a detailed description of this command.
- `concurrency` - Changes the concurrency value set in the configuration file. See Setting Concurrency With `concurrency` for a detailed description of this command.
- `excludeList` - Specifies or changes the exclude list set in the configuration file. See Specifying Exclude List Files With `excludeList` for a detailed description of this command.
- `keywords` - Specifies or changes the keyword values set in the configuration file. See Specifying Keywords With `keywords` for a detailed description of this command.
- `kfl` - Specifies one or more known failure lists (KFLs), as reflected in the known failure analysis (part of the HTML report type). See Specifying Keywords with `kfl`.
- `priorStatus` - Specifies or changes prior status values set in the configuration file. See Selecting Tests With `priorStatus` for a detailed description of this command.
- `tests` - Specifies or changes the tests specified in the configuration file. See Specifying Tests or Directories With `tests` for a detailed description of this command.
- `timeoutFactor` - Specifies or changes the test timeout value specified in the configuration file. See Setting Timeout With `timeoutFactor` for a detailed description of this command.

Setting Specific Configuration Values

You can use the `set` command to override a specific value in a configuration file, import a Java platform properties file (Java properties file) containing the values of multiple configuration questions, and set the value of a Properties Question in a configuration file.

Override a Specific Value

Use the `set` command to override a specific value in the current configuration (`.jti`) file.

> *javatest* ... [*initial-setup commands*] ... `-set` *question-tag-name* ... [*task-command*] ...

See About the Command-Line Examples for a description of use of > *javatest* in the example.

See Command-Line Overview for a description of the command line structure.

Import a Java Properties File

You can also use `-set -file` *input-file-name* or `-set -f` *input-file-name* to import a Java properties file containing the values of multiple configuration questions. A hand-edited configuration file can be used as an input file.

> *javatest* ... [*initial-setup-commands*] ... `-set -file` *input-file-name* ... [*task-command*] ...

The harness uses the values in the input file to override the values in the configuration. Any values in the input file that are not used in the configuration are ignored.

Values changed by the `set`command are only used for the session and override but do not change the configuration file. To change a configuration file, use the Configuration Editor provided by the harness GUI.

Set the Value of a Properties Question

In configuration (`.jti`) files that use the Properties Question type, you can use the `set`command to override but not change the values of Properties Questions. This question type requires at least three values for any setting (question key, property name, the new value).

> *javatest* ... [*initial-set-up-commands*] ... `-set` *question-tag-name property***:***value* ... [*task-command*] ...

The *question-tag-name* and *property* identify the location in the Properties Question to be changed while *value* specifies the new value for that property. If the new value is rejected by the question, the appropriate action is taken by the harness (exit with error). The error message will specify that `question-tag-name` rejected the *value* for the *property*.

Creating a Command String

When creating a command string to set specific values in a configuration, include the commands in the following sequence:

1. Include the commands required to set up a configuration.

See Setup Commands for detailed description of the available commands.

2. Include the command required to specify configuration values (`set` *question-tag-name value*).

3. (Optional) Include the `runtests` command.

See Running Tests With `runtests` for a detailed description of the command.

To use the `set`command, you must identify the *question-tag-name* associated with the *value* in the configuration file that you are changing. In the command line, following the `set`command, enter the *question-tag-name* and its new *value*:

A value can only be changed if its *tag-name* exists in the initialized configuration file. If the configuration does not include the *tag-name* you must use the Configuration Editor in the harness GUI to include the question and value in the configuration file.

See Obtaining the Question tag-name for detailed information about the *tag-name* for the question. See Formatting Configuration Values for editJTI or -set for detailed information about formatting the values. See Detailed Examples for examples of using the `set` command and the *tag-name*.

Detailed Example of Setting Test Suite Specific Values

In the following example, *myconfig*`.jti` represents a file name that might exist on your system.

Command Options Example:

```
> javatest -config myconfig.jti -set jckdate.gmtOffset 8 -runtests
```

See Formatting a Command for descriptions and examples of other command formats.

Setting Concurrency with `concurrency`

If you are running the tests on a multi-processor computer, you can use concurrency to speed your test runs. Use the `concurrency` command to specify the number of tests to run concurrently:

> *> javatest* ... [*initial-set-up commands*] ... `-concurrency` *number* ... [*task-command*] ...

See About the Command-Line Examples for a description of the use of > *javatest* in the example.

Unless your test suite restricts concurrency, the maximum *number* of threads specified by the `concurrency` command is 50. See your test suite documentation for additional information about using concurrency values greater than 1.

When creating a command string to specify the number of tests to run concurrently, include the commands in the following sequence:

1. Set up a configuration
2. Specify the concurrency value (`concurrency` *number*)
3. Include the `runtests` command (optional).

See Command-Line Overview for a description of the command line structure.

Detailed Example of `concurrency` Command

In the following example, *myconfig*`.jti` represents a file name that might exist on your system and value represents a numeric value from 1 to 50 that you might use.

Command Options Format Example:

> *> javatest* `-config` *myconfig*`.jti -concurrency` *value* `-runtests`

See Formatting a Command for descriptions and examples of other command formats.

Specifying Exclude Lists With `excludeList`

Test suites can supply exclude list files which contain the list of tests that the harness is not required to run. Exclude list files conventionally use a `.jtx` extension. Once you have set up a configuration, you can use an `excludeList` command to specify the exclude list for your test run:

> *javatest* ... [*initial-setup-commands*] ... `-excludeList` *path/filename1 path/filename2* ... [*task-command*]

For example, to specify multiple *path/filename* arguments , issue the `-excludeList` command once followed by multiple *path/filename* arguments separated by spaces. For example:

> *javatest* ... [*initial-setup-commands*] ... `-excludeList aaaa.jtx bbb.jtx C:\myconfig\cc.jtx` ... [*taskcommand*]

Note, because a space is a separator, file path arguments cannot contain spaces (for example, `C:\Program Files\myconfig\foobar` will not work).

See About the Command-Line Examples for a description of the use of > *javatest*.

See Command-Line Overview for a detailed description of the command-line structure.

The exclude list that you specify in the command line overrides any exclude list specified in the configuration file without changing the configuration file. To specify an exclude list, include the commands in the following sequence:

1. Include the commands required to set up a configuration.See Setup Commands for a description of the commands.

2. Include the commands to specify an exclude list (`excludeList` *path/filename*).

3. (Optional) Include a task command (such as `runtests`).

   See Task Commands for the commands that you can include.

Detailed Example of `excludeList` Command

In the following example, *myconfig*`.jti` and *myexcludelist*`.jtx` represent file names that might exist on your system.

Command Options Format Example:

```
> javatest -config myconfig.jti -excludeList myexcludelist.jtx -
runtests
```

See Formatting a Command for descriptions and examples of other command formats that you can use.

Specifying Keywords With `keywords`

The test suite may provide keywords that you can use on the command line to restrict the set of tests to be run. Use the `keyword` command to specify the keywords used to filter the tests that are run.

```
> javatest ... [initial-setup-commands] ... -keywords expression ... [task-command] ...
```

See About the Command-Line Examples for a description of the use of > *javatest* in the following example.

See Command-Line Overview for a detailed description of the command line structure.

Refer to the test suite documentation for a list of supported keyword expressions and logical operators.

When creating a command string that specifies keywords, include the commands in the following sequence:

1. Include the commands required to set up a configuration.

See Set-up Commands for a description of the commands.

2. Include the commands to specify keywords used (`keywords` *expression*).
3. (Optional) Include a task command (such as `runtests`).

See Task Commands for the commands that you can include.

Detailed Example of `keywords` Command

In the following example, *myconfig*`.jti` and *myexcludelist*`.jtx` represent file names that might exist on your system.

Command Options Format Example:

```
> javatest -config myconfig.jti -keywords interactive -runtests
```

See Formatting a Command for descriptions and examples of other command formats that you can use.

Specifying Known Failures Lists With `kfl`

The ability to specify a known failures list is enabled in the configuration editor interview, as described in Creating Reports in the *Graphical User Interface User's Guide*. In the user interface, if the value for "Specify a Known Failures List?" is Yes and you have specified KFLs, they become the default values, and are used when you create reports using Reports > Create New Report or using the `-writeReports` HTML report type.

The `-kfl` option enables you to change the default list of KFL files from the command line.

You can call the KFL file(s) as follows. Multiple files are separated by spaces:

```
java -jar ... -kfl foo.kfl bar.kfl path/foobar.kfl -runtests
```

Note – Note, because a space is a separator, file path arguments cannot contain spaces (for example, `C:\Program Files\myconfig\foobar` will not work).

See Command-Line Overview for a detailed description of the command line structure. See Formatting a Command for descriptions and examples of other command formats that you can use.

Selecting Tests With `priorStatus`

Tests can be selected for a test run based on their prior test status. Use the `priorStatus` command to run tests based on their results from a previous test run:

> *javatest* ... [*initial-set-up commands*] ... `-priorStatus fail,error` ... [*task-command*] ...

See About the Command-Line Examples for a description of the use of > *javatest* in the example.

The *status-arguments* that can be used are pass, fail, error, and notRun. If you use more than one argument, each argument must be separated by a comma.

When creating a command string to specify the prior test status, include the commands in the following sequence:

1. Set up a configuration.
2. Specify the prior test status (`priorStatus` *status-arguments*).
3. Include a Task Command such as `runtests` (optional).

See Command-Line Overview for a detailed description of the command-line structure.

Detailed Example of `priorStatus` Command

In the following example, *myconfig*`.jti` represents a configuration file name that might exist on your system.

Command Options Format Example:

> *javatest* `-config` *myconfig*`.jti -priorStatus fail,error -runtests`

See Formatting a Command for descriptions and examples of other command formats that you can use.

Specifying Tests or Directories With `tests`

You can specify one or more individual tests or directories of tests for the harness to run. The harness walks the test tree starting with the sub-branches or tests you specify (or both) and executes all tests that it finds, excluding tests that are filtered out.

You can use the `tests` command to specify one or more individual tests or directories of tests:

`tests` *path/filename*

See Command-Line Overview for a description of the command line structure.

When creating a command string, include the commands in the following sequence:

1. Include the commands required to set up a configuration.

See Setup Commands for detailed description of the available commands.

2. Include the commands required to specify tests or directories of tests (`tests` *path/filename*).

3. (Optional) Include a task command such as the `runtests` command.

See Task Commands for a description of the available commands.

Example of `tests` Command

In the following example, *path/filename* represents a file name that might exist on your system.

Command Options Format Example:

> *javatest* ... [*initial-setup-commands*] ... `-tests` *path/filename* ... [*task-command*] ...

See Formatting a Command for descriptions and examples of other command formats that you can use.

See About the Command-Line Examples for a description of the use of > *javatest* in the example.

Setting Timeout With `timeoutFactor`

Each test in a test suite has a timeout limit. The harness waits for a test to complete for the duration of that limit before moving on to the next test. You can use the `timeoutFactor` command to change the timeout limit:

> *javatest* ... [*initial-setup-commands*] ... `-timeoutFactor` *number* ... [*task-command*] ...

See About the Command-Line Examples for a description of the use of > *javatest* in the example.

See Command-Line Overview for a detailed description of the command-line structure.

Each test's timeout limit is multiplied by the time factor value. For example, if you specify a value of 2.0, the timeout limit for tests with a 10 basic time limit becomes 20 minutes. See Formatting a Command for descriptions of the command formats. Note that the format of the value input for the timeout factor is dependant on the locale.

When creating a command string to change the timeout limit, include the commands in the following sequence:

1. Include the commands required to set up a configuration.

See Setup Commands for detailed description of the available commands.

2. Include the commands required to specify the timeout limit (`timeoutFactor` *number*).

3. (Optional) Include the `runtests` command.

See Running Tests With `runtests` for a detailed description of the command.

Detailed Example of `timeFactor` Command

In the following example, *myconfig*`.jti` and *myexcludelist*`.jtx` represent file names that might exist on your system.

Command Options Format Example:

```
> javatest -config myconfig.jti -timeoutFactor 2.0 -runtests
```

See Formatting a Command for descriptions and examples of other command formats that you can use.

Additional Setup Commands

In most cases, you use the command line to perform functions that are also available through the GUI. However, you can also use the command line to specify how the harness starts.

When starting the harness, you can include additional commands in the command line to:

- Include all system properties in test execution environments.
- Set an environment variable that you want inherited in every test environment.

- Set the agent pool port number.
- Set the agent pool timeout.
- Start the active agent pool.

The harness uses a new desktop when you include GUI commands in the command line.

The following table describes the commands used in the command line to specify how the harness starts.

TABLE 4 JavaTest Harness Commands

Command	Function
`-EsysProps`	Includes all system properties in test execution environments.
`-E`*name*=*value*	Sets an environment variable that is inherited in every test environment created. The `-E`*name*=*value* command tunnels in values from the external shell. The method used in previous versions of the harness to tunnel in values from the external shell is now deprecated.
`-agentPoolPort` *port*	Set the agent pool port number. Use this command only when you are configuring the harness and the agent to use a port other than 1907.
`-agentPoolTimeout` *#seconds*	Set the agent pool timeout. Sets the number of seconds that the harness waits between tests for an available agent before reporting the test result as an error. The default value of 180 seconds is usually sufficient. You can also set this value in the GUI if you are not running the harness from the command line.
`-startAgentPool`	Start the active agent pool. If you use an active agent and run the harness from the command line, you must add `-startAgentPool`to the command string to start the agent pool.

5

Task Commands

In the command line, after setting up a configuration, you can include commands to perform tasks such as run tests, and write reports. See Set-up Commands for detailed information about setting up a configuration.

Information about the following task commands can be found in the following topics:

- Running Tests With runtests
- Writing Reports

Running Tests With `runtests`

Use the `runtests` command to run the tests specified in the configuration.

> *> javatest* [*monitor-option*] [*setup-commands*] ... `-runtests` ...

See About the Command-Line Examples for a description of the use of > *javatest* in the example.

See Command-Line Overview for a detailed description of the command-line structure.

You can also use the `runtests` command as part of a sophisticated command sequence that resembles and functions as a script. You can include command files and multiple commands in the same command string to programmatically perform repeated, multiple test runs of different configurations without starting the harness GUI.

See Using Command Files for detailed information about creating and using command files.

A *monitor-option* can be set in the command line to display test progress information during the test run. See Monitor Test Progress Option for detailed information about setting this option.

When creating the command string to run one or more tests, include the commands in the following sequence:

1. (Optional) Include the command required to monitor a test run.

See Monitoring Test Progress With `verbose` for detailed information about the command.

2. Include the commands required to set up a configuration.

See Setup Commands for detailed description of the available commands.

3. Include the `runtests` command.

Detailed Example of `runtests` Command

In the following example, *myconfig*`.jti` represents a configuration file name that might exist on your system.

Command Options format example:

```
> javatest -config myconfig.jti -runtests
```

See Formatting a Command for descriptions and examples of other command formats that you can use.

Monitoring Test Progress With `verbose`

Including the `verbose` command and optional monitoring options in a run command allows the user to monitor test progress from the command line. This command uses `stdout` to display the specified levels of monitoring test run progress. This monitoring function is not available in the GUI.If you use the `verbose` command and options, set it as the first flag in the command line. Because it takes effect at the point in the command line where it appears, if the `verbose` command does not preceed the other commands, commands executed before it appears on the command line are not be shown.

Monitoring Options

The monitoring options are specified in the command line as a comma-separated list following the `-verbose` option. A colon (:) is used to separate the `-verbose` command from the options. Ordering and capitalization within the list are ignored. Whitespace within the list is prohibited.

If you do not specify a level, the `progress` option is automatically used.

> *javatest* `-verbose:` *monitor-option* [*setup-commands*] ... `-runtests` ...

See About the Command-Line Examples for a description of the use of > *javatest* in the example.

See Examples of Monitoring Output for detailed examples of the command line.

The following table describes monitoring options specified in the command line.

TABLE 5 Monitoring Options

Option	Description
`commands`	Traces the individual harness commands as they are executed. If this option is used, it should be set first in the command line. Traced harness commands include options given on the command line, commands given in command strings, and commands given in command files.
`no-date`	Does not prefix entries with the data and time stamp. Normally, each logical line of output prints the month, day, hour, minute and second.
`non-pass`	Prints non-passing (error, fail) test names and their status string. The status string includes the status (error, fail) and the reason for the failure or error.
`max`	Outputs the maximum possible amount of output. This includes all the options which are individually available. If this option is present, only the `no-date` and `quiet` flags have any additional effect.
`quiet`	Suppresses any output from the verbose system. It might be useful to temporarily deactivate monitoring while debugging, without removing other levels coded into a script. `-verbose:stop,progress,quiet` results in no output, as does `-verbose:quiet,stop,progress`. This option takes precedence over all other options. It does not suppress the pass, fail, and error statistics printed at the end of the test run.
`start`	Prints the test name when it goes into the harness' engine for execution. Note: On some test suites, this might only indicate that the test has been handed to the plug-in framework, not that it is actually executing.
`stop`	Prints the test name and status string (see `non-pass`) when a test result is received by the harness.
`progress`	Prints a progress summary, which indicates pass, fail, error, and not-run numbers. If any of the `max`, `non-pass`, `stop`, or `stop` options were specified, each summary migh be printed on its own line. If not, the summary will be updated on the current line. The progress information is printed/updated each time a test result is reported to the harness.

Detailed Examples of Monitoring Commands

The following are seven examples of monitoring commands and their resulting command line output:

- An example of the default monitoring output:

```
> java -jar lib/javatest.jar -verbose -open foo.jti -runtests
  14:21:31 Sept 14 - Harness starting test run with configuration
"foo".
  14:24:33 Sept 14 - Pass: 12  Fail: 0  Error: 1  Not-Run: 33
  14:24:30 Sept 14 - Finished executing all tests, wait for cleanup...
  14:26:31 Sept 14 - Harness finished test run.
```

- An example of the `start` monitoring output:

```
> java -jar lib/javatest.jar -verbose:start -open foo.jti -runtests
  14:21:31 Sept 14 - Harness starting test run with configuration
"foo".
  14:24:39 Sept 14 - Running foo/bar/index#id1
  14:24:30 Sept 14 - Test run stopped, due to failures, errors, user
request. Wait for cleanup...
  14:26:31 Sept 14 - Harness finished test run.
```

- An example of the `start` and `stop` monitoring output:

```
> java -jar lib/javatest.jar -verbose:start,stop -open foo.jti -
runtests
  14:21:31 Sept 14 - Harness starting test run with configuration
"foo".
  14:24:31 Sept 14 - Running foo/bar/index#id1
  14:24:32 Sept 14 - Finished foo/bar/index#id1 Fail.  Invalid index
did not throw exception.
  14:26:33 Sept 14 - Running foo/bar/index#id2
  14:27:34 Sept 14 - Finished foo/bar/index#id2 Pass.
  14:28:35 Sept 14 - Running foo/bar/index#id3
  14:29:36 Sept 14 - Finished foo/bar/index#id3 Error.  Cannot invoke
JVM.
  14:30:30 Sept 14 - Finished executing all tests, wait for cleanup...
  14:30:31 Sept 14 - Harness finished test run.
```

- An example of the `no-date`, `start`, and `stop` monitoring output:

```
> java -jar lib/javatest.jar -verbose:no-date,start,stop -open
foo.jti -runtests
```

```
  Harness starting test run with configuration "foo".
  Running foo/bar/index#id1
  Finished foo/bar/index#id1 Fail.  Invalid index did not throw
exception.
  Running foo/bar/index#id2
  Finished foo/bar/index#id2 Pass.
  Running foo/bar/index#id3
  Finished foo/bar/index#id3 Error.  Cannot invoke JVM.
  Test run stopped, due to failures, errors, user request. Wait for
cleanup...
  Harness finished test run.
```

- An example of the `non-pass` monitoring output:

```
> java -jar lib/javatest.jar -verbose:non-pass -open foo.jti -
runtests
  Harness starting test run with configuration "foo".
  Running foo/bar/index#id1
  Finished foo/bar/index#id1 Fail.  Invalid index did not throw
exception.
  Running foo/bar/index#id2
  Finished foo/bar/index#id2 Pass.
  Test run stopped, due to failures, errors, user request. Wait for
cleanup...
  Harness finished test run.
```

- An example of the `progress` and `non-pass` monitoring output:

```
> java -jar lib/javatest.jar -verbose:progress,non-pass -open foo.jti
-runtests
  14:23:39 Sept 14 - Harness starting test run with configuration
"foo".
  14:24:39 Sept 14 - Pass: 12  Fail: 0  Error: 0  Not-Run: 33
  14:25:32 Sept 14 - Finished foo/bar/index#id1 Fail.  Invalid index
did not throw exception.
  14:26:39 Sept 14 - Pass: 12  Fail: 1  Error: 0  Not-Run: 32
  14:27:39 Sept 14 - Pass: 12  Fail: 1  Error: 0  Not-Run: 32
  14:30:36 Sept 14 - Finished foo/bar/index#id3 Error.  Cannot invoke
JVM.
  14:32:39 Sept 14 - Pass: 12  Fail: 1  Error: 1  Not-Run: 31
  14:33:01 Sept 14 - Test run stopped, due to failures, errors, user
request. Wait for cleanup...
  14:33:10 Sept 14 - Harness finished test run.
```

- An example of the `no-date` and `max` monitoring output:

```
> java -jar lib/javatest.jar -verbose:no-date,max -open foo.jti -
runtests
  Harness starting test run with configuration "foo".
  Running foo/bar/index#id1
  Finished foo/bar/index#id1 Fail.  Invalid index did not throw
exception.
  Pass: 0  Fail: 1  Error: 0  Not-Run: 33
  Running foo/bar/index#id2
  Finished foo/bar/index#id2 Pass.
  Pass: 1  Fail: 1  Error: 0  Not-Run: 32
  Test run stopped, due to failures, errors, user request. Wait for
cleanup...
  Harness finished test run.
```

Using the `batch` Command

The `batch` command is a legacy command that is used to run tests from the command line or as part of a build process. If a task command is not included in the command line, the harness begins running tests automatically. The `runTests` command supercedes the `batch` command.

The `batch` command is also used in the command line to close the harness when all commands are processed. If the `batch`command is used, the harness GUI will not start unless explicitly started by another commands in the command string.

If the GUI is started in batch mode (such as including the `monitorAgent` command), after all commands are executed the harness displays a dialog that enables the user to cancel the automatic shutdown and to use the harness GUI in normal mode.

In its legacy format, the `batch` command was required to precede the other commands. In the present format, the `batch` command can be specified in any location on the command line.

> *javatest* ... `-batch` ... [*setup-commands*] ... [*task-command*] ...

See About the Command-Line Examples for a description of use of > *harness* in the example.

See Command-Line Overview for a description of the command-line structure.

Detailed Example of `batch` Command

In the following example, *myconfig*`.jti` represents a configuration file name that might exist on your system.

Command Options Format Example:

```
> javatest -batch -config myconfig.jti
```

See Formatting a Command for descriptions and examples of other command formats that you can use.

Using the `observer` Command

The `observer` command is an advanced command that allows you to register `com.sun.javatest.Harness.Observer` for monitoring a test run. For example, an observer can monitor the progress of each test run and implement custom behavior such as sending an email message if a test in a test run fails. See the API documentation for details or contact the harness team for help in using an observer.

Writing Reports With `writeReport`

Use the `-writeReport` command with the `-type` and `-filter` options in the command line as a separate command or as part of a series of task commands (such as run tests). Use a web browser to view the reports.

Because the harness executes commands in their command-line sequence, you must identify the work directory before the `-writeReport` command and provide the report directory as an option at the end of the command:

```
> javatest ... -workdir my-work-dir -writeReport -type
 report-type -filter report-filter my-report-dir
```

See About the Command-Line Examples for a description of the use of > *javatest* in the example.

See Command-Line Overview for a detailed description of the command line structure.

Using the `-type` Option

Include the `-type` option and a report-type argument in the command line to specify the format of the generated report. There is a direct relationship between the names of the directories in the report directories and the report type names used in the command, as shown in the following table:

TABLE 6 accessing Online Information from the GUI

Report	**Type**
HTML Report	`html`
Plain Text Report	`txt`
XML Report	`xml`
COF Report	`cof`

When the `-type` option is not used, the harness uses the default report types last used in the GUI, or `html` and `txt` if a type was not previously set in the GUI.

The harness provides a set of standard format types (`html`, `txt`, `xml` and `cof`) that you can use. In addition to the standard arguments, your test suite might provide additional custom formats. If you are unsure of the additional formats provided by your test suite, select Report > Create New Report from the GUI to display the list of available report formats.

For more on the standard report formats, see Creating Reports in the *Graphical User Interface User's Guide*.

Using the `-filter` Option

When the `-filter` option is not used, the harness uses the default setting of `currentConfig`. Include the `-filter` and a filter option (`lastRun`, `currentConfig`, or `allTests`) in the command line to specify the filter used to select the test results that are reported.

Use the `lastRun` filter option (corresponds to Last Test Run in the GUI) to select test results status for all folders and tests included in the last test run even if you have exited the harness since the last test run.

Use the `currentConfig` filter option (corresponds to Current Configuration in the GUI) to select the folders and test results status specified by the current configuration.

Use the `allTests` filter option (corresponds to All Tests in the GUI) to select test results status for all tests in the work directory, including any tests that were excluded from the last test run.

Detailed Example of `writeReport` Command

In the following example, *my-work-dir* represents a work directory name that might exist on your system, and two known failure lists are specified.

Command Options Format Example:

> *> javatest* -workdirectory *myworkdirectory* `-config foo.jti -kfl foo.kfl -writeReport -type html -filter allTests` *myReportDir*

See Formatting a Command for descriptions and examples of other command formats that you can use.

6

Desktop Options and Preferences

In most cases, command-line options perform functions that are also available through the GUI. However, there are several situations in which using command-line options to specify how the harness starts are uniquely useful or necessary.

When starting the harness you can use options in the command line to perform the following tasks:

- Use a new desktop when starting the harness GUI.
- Restore tools state when starting the harness GUI.
- Specify a look and feel when starting the harness GUI.
- Change the status colors when starting the harness GUI.

Startup Commands

When starting the harness, include `-newDesktop` in the command string to start the harness GUI without using a previous desktop. The harness ignores any previous desktop information and opens the Quick Start wizard.

Note: Some test suites do not implement the optional Quick Start wizard. It won't be available if the test suite architect disabled it.

Note – The harness uses a new desktop when you include GUI options in the command line. Using this option preserves any preferences set for the desktop. Use the following example to start the harness with a new Desktop.

```
> javatest -newDesktop
```

See About the Command-Line Examples for a description of the use of > *javatest*.

Restore Tools State

Specify `-resume` to restore the last-saved tools state. Tools settings will be restored, even if the preference Check Restore Tools State on Start was disabled from the user interface.

Specifying Look and Feel

The `-laf` option selects the look and feel for a JavaTest session. This option affects the GUI only. It has no effect if the harness is used solely as a command line tool.

The look and feel option `-laf` requires one of the following arguments:

- `default`: Use the harness default. The Java look and feel options are `nimbus` or `metal`. Images in the help and documentation depict these styles.
- `nimbus`: The modern Java look and feel. This is the harness default if it is available in the runtime.
  http://download.oracle.com/javase/6/docs/technotes/guides/jweb/otherFeatures/nimbus_laf.html
- `metal`: The legacy Java look and feel used through version 4.4 of the harness.
  http://download.oracle.com/javase/6/docs/api/javax/swing/plaf/metal/MetalLookAndFeel.html
- `sys` or `system`: Use the system-default look and feel that matches your desktop style (`nimbus` or `metal`). To achieve the best results from some features of your operating system, such as high contrast color schemes, large fonts and other accessiblity related functions, it might be necessary to select the `system` look and feel.

A sample command would be:

> *javatest* `-laf metal`

To use the native look and feel specify the `system` option. For example:

```
> java -jar javatest.jar -laf system ...
```

Specifying Status Colors

The harness enables you to specify the status colors used in the GUI. This property is set on the command line as a system property when starting the harness GUI. Status colors set this way are added to the user preferences and restored in subsequent sessions.

The user can specify each status color by declaring system properties in the following format:

`-Djavatest.color.passed=`*color-value* ...

`-Djavatest.color.failed=`*color-value* ...

`-Djavatest.color.error=`*color-value* ...

`-Djavatest.color.notrun=`*color-value* ...

`-Djavatest.color.filter=`*color-value* ...

The *color-value* used must be an RGB value parsable by the `java.awt.Color` class (octal, decimal, or hex).

The value portion of the color property must be explicitly defined. The value portion of the property accepts hex values, prefixed by either a pound character (`#`) or a zero-x (`0x`).

Values can also be specified in octal, in which case the value begins with a leading zero and must be two or more digits.

The following are possible formats for setting color integers:
`#ffaa66` (hex)
`0xffaa66` (hex)
`0111177` (octal)

This is a detailed example of specifying a status color. You might have to escape the pound character for the command to work on your platform.

```
> java -Djavatest.color.passed=#00FF00 -jar
[jt_dir/lib/]javatest.jar
```

7

Information Commands

The harness command-line interface enables you to display command-line help, online help, or version information without starting the harness.

> *> javatest* [*Information Command*]

Including an information command at the end of the command line causes the harness to display harness information without starting the GUI.

The following topics provide detailed information about the commands that can be used to display harness information:

- Displaying Harness Command-Line Help
- Displaying Harness Online Help
- Display Harness Version Information

Command-Line Help

The harness allows you to display the command-line interface in the following forms:

- All information
- Topic information
- Word search information

Displaying All Information

To display all of the information in command-line help, include `-help all` at the end of the command line.

```
> javatest -help all
```

See About the Command-Line Examples for a description of the use of > *javatest*.

Displaying Topic Information

To display only the command-line help for specific topics, include `-help` and the name of the topic at the end of the command line.

```
> javatest -help topic name
```

See About the Command-Line Examples for a description of the use of > *harness*.

The following table lists the available command-line help topics.

TABLE 7 Options Used to Display Command-Line Information

Topic	Function
Desktop	Displays information about the command-line options for starting the harness graphical user interface.
harness Agent	Displays information about the command-line options for the harness Agent.
Batch Mode	Displays information about the command-line options for running tests in batch mode.
Configuration	Displays information about the command-line options for setting up and changing a configuration.
Environment	Displays information about the command-line options for adding values into harness environments.
HTTP server	Displays information about the command-line options for the harness HTTP server.
Options	Displays information about the command-line options accepted on the command line.
Files	Displays information about the types of files accepted as command-line arguments.

Display the List of Available Topics

To display the list of help topics provided by command-line help, include `-help`, `-usage`, or `-?` at the end of the command line.

```
> javatest -help
```

See About the Command-Line Examples for a description of the use of > *harness*.

Searching for Words and Phrases

The harness allows you to search the full command-line help for a specific word or phrase and then displays only the information containing that word or phrase.

To display command-line help information containing a specific word or phrase, include `-help` and the word or phrase at the end of the command line.

> *> javatest* `-help` *word or phrase*

See About the Command-Line Examples for a description of the use of > *javatest*.

Example: *> javatest* `-help services` *test*

The console displays the following output:

```
Services        Options for the JavaTest Harness test suite services
management
   -startServices  Change the service startup policy, possible values
are
                     [lazy, up_front, manually].
```

For complete details and examples, see the harness online help. You can access help directly from the command line with `-onlineHelp` <*word*>, or you can start the harness and use the Help menu. The online help is also available in PDF format in the harness documentation directory.

Displaying Online Help

Without opening the GUI, the harness enables you to display the complete online help. To display the online help, include the `-onlinehelp` command at the end of the command line.

> *> javatest* `-onlinehelp`

See About the Command-Line Examples for a description of the use of > *javatest*.

Displaying Harness Version Information With `version`

To display the version, location, and build information of the installed copy of the harness, include the `-version` command at the end of the command line.

```
> javatest -version
```

See About the Command-Line Examples for a description of the use of > *javatest*.

8

Legacy Commands

The command-line interface supports the commands used in previous versions of the harness. In most cases, using the current commands is preferred, however, if you are running a test suite that uses a parameter file you can continue to use options in the command line to specify parameter values. These commands are deprecated and might be removed from future versions of the harness.

See Using Parameter Commands for a detailed description of these Legacy Commands.

Using Deprecated Parameter Commands

If you are running a test suite that does not use a parameter file, use the current commands instead of the `-params` command and option.

If you are running a test suite that uses a parameter file (`.jtp`), you can specify different parameter values by including `-params` and the appropriate parameter command in the command line.

> javatest ... `-params` [*command*] [*value*] [*task-commands*] ...

See About the Command-Line Examples for a description of the use of > *javatest*.

The following table describes the parameter commands.

TABLE 8 Parameter Commands

Command	Description
`-t` *testsuite* or `-testsuite` *testsuite*	Specifies the test suite.
`-keywords` *keyword-expr*	Restricts the set of tests to be run based on keywords associated with tests in the test suite.
`-status` *status-expr*	Includes or excludes tests from a test run based on their status from a previous test run. Valid status expressions are error, failed, not run, and passed.
`-excludeList` *exclude-list-file*	Specifies an exclude list file. Exclude list files contain a list of tests that are not to be run. Exclude list files conventionally use the `.jtx` extension and are normally supplied with a test suite.
`-envFile` *environment-file*	Specifies an environment file that contains information used by the harness to run tests in your computing environment. You can specify an environment file for the harness to use when running tests.
`-env` *environment*	Specifies a test environment from an environment file.
`-concurrency` *number*	Specifies the number of tests run concurrently. If you are running the tests on a multi-processor computer, concurrency can speed your test runs.
`-timeoutFactor` *number*	Increases the timeout limit by specifying a value in the time factor option. The timeout limit is the amount of time that the harness waits for a test to complete before moving on to the next test. Each test's timeout limit is multiplied by the time factor value. For example, if you specify a value of 2, the timeout limit for tests with a 10 basic time limit becomes 20 minutes.
`-r` *report-directory* or `-report` *report-directory*	Specifies the directory where the harness writes test report files. If this path is not specified, the reports are written to a directory named report in the directory from which you started the harness.
`-w` *work-directory* or `-workDir` *workDirectory*	Specifies a work directory for the test run. Each work directory is associated with a test suite and stores its test result files in a cache.

9

The Lite Harness

Topics

Overview

The lite harness (`jtlite.jar`) can run test suites on devices that have either of two limitations:

- The Java implementation on the test device has no network access, as can be the case with an embedded device. The test harness, running on a laptop, cannot work in such a networkless environment because it uses an agent to run tests on the device, and the agent requires a network to communicate with the harness. By contrast, the lite harness runs entirely on the test device, executing tests there without an agent and without dependencies on networking APIs.
- The Java implementation on which the harness runs supports a subset of Java SE APIs. To accommodate API limitations, the lite harness has no graphical user interface and supports only a subset of the test harness's CLI commands.

Supported Commands

The lite harness supports all test harness commands and options, except the following:

- -newDesktop
- -resume
- -laf
- -onlineHelp
- -type xml (suboption of `-writeReport`)
- -type cof (suboption `-writeReport`)

Device Requirements

- The device on which the lite harness runs must have a Java runtime environment that implements at least the APIs defined as the JDK8 compact1 profile in http://cr.openjdk.java.net/~mr/se/8/java-se-8-edr-spec.01.html#s8. An example is the headless edition of Oracle Java SE Embedded 7, available from http://www.oracle.com/technetwork/java/embedded/downloads/javase/index.html.
- You must be able to run the `java` launcher command (version 6 or later) on the test device. You can do this with a keyboard and monitor connected to the device or over a operating system-level network connection that supports a remote protocol such as `ssh`. Run `java -version` to verify the Java installation.
- You must be able to transfer files to the device. Example transfer mechanisms include:
- A operating system-level network connection to the device that supports remote transfer commands such as `scp` or `sftp`.
- An SD card, USB card, or other portable file system medium.
- A tool that can burn files into non-volatile memory.
- A web browser that can download files from a web server that you control.
- The device must have sufficient storage for the Java runtime, the lite harness implementation (`jtlite.jar`), the test suite to be executed, a work directory, and reports. The `jtlite.jar` file requires about 1MB of space. For Java runtime and test suite storage requirements, consult the corresponding documentation. Work directory and report sizes depend on the tests executed and reports generated.

- The device must have sufficient dynamic memory to run the lite harness performing the commands you specify. Dynamic memory requirements are so situation-specific, they can only be determined by experimentation.

Although operating system-level networking support on the device is convenient for transfering files and running the lite harness remotely, the lite harness does not have any dependencies on networking APIs or services.

Installing

To install the lite harness on the test device, transfer a copy of *javaTestInstallDir*`/lib/jtlite.jar` to the test device's file system.

You must also install the test suite on the device. Consult the test suite documentation for instructions.

Launching

Assuming that the current directory contains the `jtlite.jar` you installed, and the device's `java` launcher command can be invoked with its unqualified name, the general form of the lite harness launch command is:

```
> java -jar jtlite.jar  \
-config yourConfig.jti \
-workdir yourWorkDir \
[more commands ...] \
-runTests \
-writeReport OutputReportDir
```

In this generic example, the \ line continuation character visually separates the command line arguments for clarity. This command line selects an existing configuration (file), an existing work directory and asks the harness to run the tests and write the default report(s). The location of the test suite is implied and will be determined from hints in the work directory. For more examples, refer to Extended Command-Line Examples, substituting the lite harness's JAR file name.

Creating a Configuration File

When you use the lite harness, you must specify the name of a configuration file on the command line with `-config` *yourConfig*`.jti`. The easiest and most reliable way to create and edit a configuration file is with the test harness graphical configuration editor running on a desktop or laptop computer. When you have created or updated a configuration file, copy it to the test device's file system so you can specify it when you launch the lite harness.

Working with Reports and Test Results

It may not be efficient to analyze results on the test platform itself - users have the option of relocating results to a more powerful platform. By copying work directories or reports from the test device to your desktop or laptop computer, you can examine and manipulate them with the test harness and other tools. For instance, you can use the report converter to merge reports or produce XML, and you can browse test information with the test harness GUI.

10

Troubleshooting

The harness provides information in the following topics that you can use to troubleshoot problems:

- Exit Codes
- Harness Fails During Use
- Problems Running Tests
- Problems Writing Reports
- Problems Moving Reports
- Numeric Output Mangled (on Windows platforms)

Exit Codes

When the harness exits, it displays an exit code that you can use to determine the exit state. The following table contains a detailed description of the exit codes.

TABLE 9 JavaTest Harness Exit Codes

Exit Code	Description
0	If tests were executed, all tests had passed results.
1	One or more tests were executed and had failed results.
2	One or more tests were executed and had errors.
3	A problem exists with the command-line arguments.
4	Harness internal error exists.

Harness Fails During Use

If the harness fails, you can use the `harness.trace` file in your work directory to troubleshoot the problem. The `harness.trace` file is a plain-text file that contains a log of harness activities during the test run. It is written in the work directory, is incrementally updated, and is intended primarily as a log of harness activity.

Problems Running Tests

The goal of a test run is for all tests in the test suite that are not filtered out to have passing results.

If the root test suite folder contains tests with errors or failing results, you must troubleshoot and correct the cause to successfully complete the test run. See Troubleshooting With the GUI in the *Graphical User Interface User's Guide* for information about the resources that the harness provides for troubleshooting.

Tests With Errors

Tests with errors are tests that could not be executed by the harness. These errors usually occur because the test environment is not properly configured. Use the GUI Test tabbed panes and configuration editor window to help determine the change required in the configuration. See Troubleshooting With the GUI in the *Graphical User Interface User's Guide* for information about the resources that the harness provides for troubleshooting.

Tests That Fail

Tests that fail are tests that were executed but had failing results. The test or the implementation may have errors.

Use the GUI Test Manager tabbed panes to identify and correct a test failure. See Troubleshooting With the GUI in the *Graphical User Interface User's Guide* for information about the resources that the harness provides for troubleshooting.

Problems Viewing Reports

The harness does not automatically generate reports of test results after a test run. You must generate test reports either from the command line or from the GUI.

Problems Writing Reports

You use filters to write test reports for a specific set of test criteria. Verify that you are using the appropriate filter to generate reports of test results. See Creating Reports in the *Graphical User Interface User's Guide*.

Problems Moving Reports

Test reports contain relative and fixed links to other files that may be broken when you move reports to other directories.

You must update these links when moving reports to other directories. The harness provides an EditLinks utility that updates the links in the reports for you when moving reports.

Numeric Output Mangled (on Windows platforms)

Some users experience locale related problems when seeing numeric output on the console, for example, under a Russian locale users have observed this output problem:

```
Pass: 298 Fail: o Error: 0 Not-Run: 50a947
```

which really should be (space for every thousand):

```
Pass: 298 Fail: o Error: 0 Not-Run: 50 947
```

And in a US locale would be:

```
Pass: 298 Fail: o Error: 0 Not-Run: 50,947
```

The problem is rooted in the terminal's ability to correctly match and display the extended characters which the JVM is producing based on the locale. On Windows, the provided MS-DOS box and Terminal/Console program default to the CP866 (Cyrillic) codepage which has poor compatibility with the Unicode output which the JVM is producing.

Users can workaround this problem by explicitly changing the codepage to one compatible with the Unicode characters for the Cyrillic locale. For the Russian locale example:

```
> chcp 1251
```

Or users can force the JVM to use the default codepage for the terminal:

```
java -Dfile.encoding=cp866 -jar ...
```

These problems have not been reported on other operating systems, but could definitely occur depending on the locale being used and the terminal's ability to display the character set. The platform may also not have adequate character sets installed. Users may also observe this same problem whenever internationalize output it produced by the JVM - a similar problem could easily occur while printing dates or floating point numbers.

11

Utilities

The harness allows you to use additional utilities to remotely monitor and control a test run, browse result (`.jtr`) files without starting the harness, browse exclude list files without starting the harness, change responses in a configuration file without starting the harness, and move test reports.

Information about using utilities to perform tasks can be found in the following topics:

- Monitoring Results With HTTP Server
- Browsing `.jtr` Result Files
- Browsing Exclude List Files
- Changing Configuration Values With `EditJTI`
- Changing Configuration Values With Text Editors
- Moving Test Reports

Monitoring Results With HTTP Server

The harness provides a web server that you can use to remotely monitor and control a test run. The HTTP Server provides the following two types of output:

- HTML-Formatted Output for users to remotely monitor batch mode test runs in a web browser.
- Plain Text Output for use by automated testing frameworks.

HTML-Formatted Output

The HTML-formatted output is provided as human-readable pages (these pages are subject to change in future releases of the harness), enabling users to remotely monitor batch mode test runs in a web browser and stop test runs that are not executing as expected. The following topics describe the HTML formatted output:

- Server Index Page
- Server Harness Page
- Server Test Result Index Page
- Harness Environment Page
- Harness Interview Page
- Stop a Test Run

accessing HTTP Server HTML-Formatted Output

1. Use the following command on the command line to activate the web server.

   > *javatest* `-startHttp -runTests` *[options]*

   See About the Command-Line Examples for a description of the use of > *javatest*.

2. Copy the URL reported to the console.

   Example:
   ```
    JavaTest Harness HTTPd - Success, active on port 1903
     JT Harness HTTPd server available at
   http://129.145.162.75:1903/
   ```

3. Launch a web browser and enter or paste the URL in the browser URL field.

   Example:
   ```
    http://129.145.162.75:1903/
   ```

Displaying the HTTP Server Index Page

The root of the web server provides an index page that only lists the handlers registered with the internal web server; not all available URLs on the server. You can also display the HTTP Server Index page by including `/index.html`at the end of the URL in the browser URL field.

Example:
```
 http://129.145.162.75:1903/index.html
```

Each harness has its own handler, identified by a unique number as the second component of the URL.

Displaying HTTP Server Harness Page

When the harness is running tests, the harness page displays the following information:

- Name and location of the current test suite.
- Location of the work directory.
- Link to view the environment information provided to the harness and used in the current test run. Displays an HTML-formatted view of the current environment.
- Link to view the configuration interview used by the harness in the current test run. Displays a formatted view of the interview settings.
- Link to view the current test results. Displays the Test Result Index page.

In addition to the list of registered handlers, the page also prints the UTC/GMT date on which that page was generated (subject to the system clock on the machine which harness is running) and provides the harness version number and build date.

You can display the HTTP Server Harness page by choosing its link on the index page or by including `/harness` at the end of the URL in the browser URL field.

Example:
```
http://129.145.162.75:1903/harness
```

Displaying the HTTP Server Test Result Index Page

The Test Result Index page displays the following information:

- Work directory
- Total number of tests in the test suite

The total number of tests is also a link to view the current test results. The test results are displayed in a two-column table, by test name and status message.

You can display the Test Result Index page by choosing its link on the harness page.

Displaying the Harness Environment Page

The Harness Environment page displays the environment information provided to the harness and used in the current test run. The environment information is displayed in an HTML table and provides a view of the current settings.

You can display the Harness Environment page by choosing its link on the harness page or by including `/harness/env` at the end of the URL in the browser URL field.

Example:
`http://129.145.162.75:1903/harness/env`

Displaying the Harness Interview Page

The Harness Interview page displays the configuration interview provided to the harness and used in the current test run.

You can display the Harness Interview page by choosing its link on the harness page or by including `/harness/interview`at the end of the URL in the browser URL field.

Example:
`http://129.145.162.75:1903/harness/interview`

Using HTTP Server to Stop a Test Run

If you want to remotely terminate a test run, include `/harness/stop`at the end of the URL in the browser URL field.

Example:
`http://129.145.162.75:1903/harness/stop`

To stop the test run, you must click the STOP button on the page displayed in the browser.

Plain Text Output

The HTTP server provides plain text output that can be used for automated monitoring of the harness during test runs. The plain text output does not include HTTP headings or HTML formatting and is intended for use by automated testing frameworks, not for viewing in web browsers. Consequently, future releases of the harness will attempt to maintain the content formatting and URLs of this output.

The following two types of harness information can be accessed by automated testing frameworks:

- Version Information
- Harness Information

accessing Version Information

The HTTP Server Version page displays version information about the JavaTest Harness harness. You can display the HTTP Server Version page by choosing its link on the index page or by including `/version` at the end of the URL in the browser URL field.

Example:
```
http://129.145.162.75:1903/version
```

A dump of the version information is provided.

Example:
`<name> Harness` <*version*> Built on August 14, 2007

accessing Harness Information

The following strings access specific information about the harness:

- `/harness/text/config`

  Currently provides, in `java.util.Properties` format, the test suite name location and work directory of the current harness configuration values.

  Example:
  ```
  testsuite.path=/export/scratch/myTest
  testsuite.name=myTest
  workdir=/export/scratch/wdsc13a
  ```
- `/harness/text/tests`

  Provides in `java.util.Properties` format the initial tests used for the current test run.

  Example:
  ```
  url0=api/java_lang
  url1=api/java_util
  ```
- `/harness/text/stats`

  Provides, in `java.util.Properties` format, the current count of test results in each state (pass, fail, error, not run). Whitespace is not present in the output:

  Example:
  ```
  Passed.=0
  Failed.=151
  Error.=54
  Not_run.=1
  ```

For performance reasons, the `Not_run` number usually equals the concurrency setting in batch mode and matches the "not run" number shown in the GUI when in GUI mode (Current Configuration view filter).

- `harness/text/results`

  Provides alternating lines of test name, test status.

  Example:
  ```
   lang/FP/fpl005/fpl00506m1/fpl00506m1.html
   Error. context undefined for hardware.xFP_ExponentRanges
   lang/FP/fpl005/fpl00506m2/fpl00506m2.html
   Error. context undefined for hardware.xFP_ExponentRanges
   vm/classfmt/atr/atrnew003/atrnew00301m1/atrnew00301m1.html
   Failed. unexpected exit code: exit code 1
   vm/classfmt/atr/atrnew003/atrnew00302m1/atrnew00302m1.html
   Failed. unexpected exit code: exit code 1
   vm/classfmt/atr/atrnew003/atrnew00303m1/atrnew00303m1.html
   Failed. unexpected exit code: exit code 1
  ```

- `/harness/text/state`

  Indicates whether harness is currently running. It will return one of the following:

  ```
  running=true
   running=false
  ```

- `/harness/text/env`

  Provides, in `java.util.Properties` format, the current environment settings for the test run.

  Example:
  ```
   command.testExecute=com.sun.jck.lib.ExecJCKTestOtherJVMCmd
  /work/jdk1.3.1/bin/java -classpath $testSuiteRootDir/classes
   -Djava.security.policy=$testSuiteRootDir/lib/jck.policy
  $testExecuteClass $testExecuteArgs
  context.nativeCodeSupported=true
  description=bar
   jniTestArgs=-loadLibraryAllowed
  nativeCodeSupported=true
  platform.expectOutOfMemory=true
  ```

Changing Configuration Values With `EditJTI`

The harness provides the `EditJTI` utility for you to use in changing the values in a configuration file from the command line. You can also make changes in a configuration by specifying the appropriate `set` command (see Command-Line Summary for detailed information).

While you can use `EditJTI` to change the order of commands in a configuration file, the dependencies between questions can introduce errors into the configuration. Use the Configuration Editor in the harness GUI when making major changes in a configuration.

If your changes to a configuration introduce errors, you can use the harness GUI Configuration Editor to troubleshoot and repair the configuration.

`EditJTI` Command Format

The `EditJTI` command loads a configuration (`.jti`) file, and applies a series of changes specified on the command line. See Formatting Configuration Values for editJTI or -set for detailed information about formatting the values.

You can save the changes in the original configuration file or save the changes in a new configuration file. You can also use `EditJTI` to generate an HTML log of the questions and responses as well as write a quick summary of the questions and responses to the console. The `EditJTI` utility provides a preview mode. Configuration files are normally backed up before being overwritten.

Example:
`java -cp lib/javatest.jar com.sun.javatest.EditJTI` [*OPTIONS*] [*EDIT-COMMANDS*] *original-configuration-file*

OPTIONS

The following are the available options:

`-help`, `-usage` or `/?`

Displays a summary of the command line options.

`-classpath` *classpath* or `-cp` *classpath*

Overrides the default classpath used to load the classes for the configuration interview. The default is determined from the work directory and test suite specified in the configuration file. The new location is specified by this option.

`-log` *log-file* or `-l` *log-file*

Generates an HTML log containing the questions and responses from the configuration file. The log is generated after edits are applied.

`-out` *out-file* or `-o` *out-file*

Specifies where to write the configuration file after the edits (if any) are applied. The default setting is to overwrite the input file if the interview is edited.

`-path` or `-p`

Generates a summary to the console output stream of the sequence of questions and responses from the configuration file. The summary is generated after edits are applied.

`-preview` or `-n`

Does not write out any files, but instead, preview what would happen if this option were not specified.

`-testsuite` *test-suite* or `-ts` *test-suite*

Overrides the default location used to load the classes for the configuration interview. The default is determined from the work directory and test suite specified in the configuration file. The new location is determined from the specified test suite.

`-verbose` or `-v`

Verbose mode. As the edit commands are executed, details of the changes are written to the console output stream.

COMMANDS

Two different types of commands are supported.

tag-name=value

Sets the value for the question whose tag is *tag-name* to *value*. It is an error if the question is not found. The question must be on the current path of questions being asked by the interview. To determine the current path, use the `-path`option. See Obtaining the Question *tag-name*.

/search-string/replace-string/

Scans the path of questions being asked by the interview, looking for responses that match (contain) the search string. In such answers, replace *search-string* by *replace-string*. Note that changing the response to a question may change the subsequent questions that are asked. It is an error if no such

questions are found.
If you use /in the search string, you use some other character (instead of /) as a delimiter.
For example:
|*search-string*|*replace-string*|
Regular expressions are not currently supported in *search-string*, but may be supported in a future release.

Depending on the shell in use, quote the commands to protect characters in them from interpretation by the shell.

RETURN CODE

The following table describes the return codes generated when a program exits.

TABLE 10 Exit Return Codes

Code	Description
0	The operations were successful. the configuration file is complete and ready to use.
1	The operations were successful, but the configuration file is incomplete and is not yet ready to use for a test run.
2	A problem exists with the command-line arguments.
3	An error occurred while trying to perform the copy.

SYSTEM PROPERTIES

Two system properties are recognized.

`EditJTI.maxIndent`

Used when generating the output for the `-path` option, this property specifies the maximum length of tag name after which the output will be line-wrapped before writing the corresponding value. The default value is 32.

`EditJTI.numBackups`

Specifies how many levels of backup to keep when overwriting a configuration file. The default is 2. A value of 0 disables backups.

The following topics provide detailed information about using `EditJTI` to perform tasks:

- Changing Configuration Values

Changing Configuration Values

When using `EditJTI` to change the values in a configuration, you can use either of the following command formats:

- Use *tag=value* for direct replacement of values. You must know the *tag-name* for the question that sets the value.
- Use */old pattern/new pattern/* to replace all occurrences (strings) of an old pattern to a new pattern. This format replaces all occurrences in the file.

When using the */old pattern/new pattern/* format, the separator can be any character. However, the string should be enclosed in quotes to avoid shell problems.

```
"|/java/jdk/1.3/|/java/jck/1.4/|"
```

Note – To run the following examples of changing configuration values, you must replace *myoriginal*`.jti` with a `.jti` file name that exists on your system. Win32 users must also replace the / file separators with \ file separators to run these examples.

Example:

```
java -cp [jt_dir/lib/]javatest.jar com.sun.javatest.EditJTI -o
mynew.jti "|/java/jdk/1.3/|/java/jck/1.4/|" myoriginal.jti
```

Generating a Log of All Updates

You can use the `-l` option to generate a log of all updates to the `.jti` file which can be used later.

Example:
`java -cp` [*jt_dir* `/lib/`] `javatest.jar com.sun.javatest.EditJTI -o` *mynew*`.jti -l` *myeditlog*`.html "|/java/jdk/1.3/|/java/jck/1.4/|"` *myoriginal*`.jti`

Previewing Without Change

You can use the -n option to preview but not perform updates to the `jti` file.

Example:
`java -cp` [*jt_dir*`/lib/`] `javatest.jar com.sun.javatest.EditJTI -n -o` *mynew*`.jti -l` *myeditlog*`.html "|/java/jdk/1.3/|/java/jck/1.4/|"` *myoriginal*`.jti`

Echoing Results of Edits

You can include the `-v` option to echo results of your edit.

Example:
`java -cp` [*jt_dir* `/lib/`] `javatest.jar com.sun.javatest.EditJTI -n -v -o` *mynewl*`.jti -l` *myeditlog*`.html "|/java/jdk/1.3/|/java/jck/1.4/|"` *myoriginal*`.jti`

Showing Paths for Debugging

The `-p` option can be used to show the path during debugging. Using `-p` options in the command string displays how the path is changed by your edit.

Example:
`java -cp` [*jt_dir*`/lib/`] `javatest.jar com.sun.javatest.EditJTI -n -o mynew.jti -l` *myeditlog*`.html -p "|/java/jdk/1.3/|/java/jck/1.4/|"`*myoriginal*`.jti`

Changing Test Suites or Creating a New Interview

The following example uses the `-ts` option to create an empty interview derived from the test suite (*mytestsuite*`.ts`). Use the `-ts` option only for very simple test suites.

Example:

```
java -cp [jt_dir /lib/] javatest.jar com.sun.javatest.EditJTI -o
mynew.jti -l myeditlog.html -ts mytestsuite.ts
"|/java/jdk/1.3/|/java/jck/1.4/|" myoriginal.jti
```

If a change is made that is not in the current interview path, the interview will be invalid and the tests cannot be run.

Do not use `EditJTI` to change the interview path, but only the values on the existing path. If you are in doubt about the current interview path, open the configuration editor window in the harness GUI and use it to change the values. The configuration editor window displays the current interview path for that question name-value pair.

Changing the HTTP Port

You can use `EditJTI` to change the HTTP port and either overwrite the original configuration file or create a new configuration file.

Note – To run the following examples, you must use a `.jti` file that exists on your system and include httpPort in your current interview path. If your current interview path does not include httpPort you will not be able to change its value from the command line. To view the current interview path, open your `.jti` file in the Configuration Editor. See Obtaining the Question tag-name for detailed information about the *tag-name* for the question.

Change the HTTP Port and Overwrite Original Configuration File

The following example changes the HTTP port used when running tests and overwrites original configuration file (*myoriginal*`.jti` in this example).

```
java -cp [jt-dir /lib/] javatest.jar com.sun.javatest.EditJTI
httpPort=8081 myoriginal.jti
```

Change the HTTP Port and Create a New Configuration File

The following example changes the HTTP port used when running tests and writes the changed configuration to a new configuration file (*myoutput*`.jti` in this example). The original configuration file (*myoriginal*`.jti` in this example) remains unchanged.

```
java -cp [jt-dir /lib/] javatest.jar com.sun.javatest.EditJTI -o
myoutput.jti httpPort=8081 myoriginal.jti
```

Doing Escapes in a UNIX System Shell

The following example uses the syntax for doing escapes in a UNIX system shell. Changes to the original configuration file (*myoriginal*.`jti` in this example) are written to a new configuration file (*my-newconfig*.`jti` in this example).

In the following example, *myoriginal*.`jti` represents a configuration file name that might exist on your system. Win32 users must also replace the UNIX system file separators (\) with Windows file separators (/) to run these examples.

To change a value in the command line, use the *tag-name* for the question that sets the value. See Obtaining the Question tag-name for detailed information about viewing the *tag-name* for the question.

```
java -cp [jt-dir/lib/] javatest.jar com.sun.javatest.EditJTI
-o my-newconfig.jti test.serialport.midPort=/dev/term/a
test.connection.httpsCert="\"CN=<Somebody>, OU=<People>, O=
<Organisation>, L=<Location>, ST=<State>, C=US\""myoriginal.jti
```

12

Changing Configuration Values With Text Editors

The harness enables you to use a text editor from within a script (such as `sed`) to change responses in a configuration file and then launch the harness to run tests.

The configuration file is a standard harness properties file in which double backslashes and escaped new lines are required. If you edit this file in a text editor, you must also remove the checksum for harness to accept it when running tests.

Checksums are used by the harness to ensure that a configuration used to run tests is complete. By removing the checksum, you risk introducing errors in the configuration used to run tests.

Test your changes in the Configuration Editor before applying them in a text editor. The Configuration Editor checks the value and displays the correct set of related questions. See Configuring a Test Run in the *harness User's Guide: Graphical User Interface* for detailed information about the Configuration Editor.

The relationship between the questions in a configuration depends on the test suite and the interdependence of the questions. A change in the value of one question might change subsequent, related configuration questions and values. If your response changes the set of required configuration values, the Configuration Editor displays the incomplete configuration and provides you with a new set of required configuration questions.

After you have tested your changes and are satisfied with the results, you can use the text editor to apply them to the configuration. Remove the checksum from the configuration file before using the changed configuration to run tests.

Moving Test Reports

Test reports contain relative and fixed links to files that break when they are moved. To prevent this, the harness provides an `EditLinks` command-line utility in the main harness JAR file, `javatest.jar`, for you to use when moving test reports.

The `EditLinks` utility checks all files with names ending in `.html` for HTML links beginning with file names you specified in the `EditLinks` command. These links are rewritten using the corresponding replacement name from the `EditLinks` command and are copied to the new location. `EditLinks` copies all other files to the new location without change.

Format of the `EditLinks` Command

Example:
`java -classpath` [*jt_dir*`/lib/`] `javatest.jar`
`com.sun.javatest.EditLinks` *OPTIONS file...*

OPTIONS

The available *OPTIONS* are as follows:

`-e` *oldPrefix newPrefix*

Any links that begin with the string *oldPrefix* are rewritten to begin with *newPrefix*. Note that only the target of the link is rewritten, and not the presentation text. The edit is effectively transparent when the file is viewed in a browser. Multiple `-e` options can be given. When editing a file, the options are checked in the order they are given.
For example, if the argument

```
-e /work/ /java/jck-dev/scratch/12Jun00/jck-lab3/
```

is used on a file that contains the segment

```
<a href=
"/work/api/java_lang/results.jtr">/work/api/java_lang/re
sults.jtr</a>
```

, the following text shown in bold will match.
`<a href=`
"/work/api/java_lang/results.jtr">/work/api/java_lang/results.jtr</a>
The resulting new file will contain the following text:

```
<a href="/java/jck-dev/scratch/12Jun00/jck-
lab3/api/java_lang/results.jtr">/work/api/java_lang/resu
lts.jtr</a>
```

`-ignore` *file*

When scanning directories, ignore any entries named *file*. Multiple `-ignore` may be given.

For example, `-ignore SCCS` causes any directories named SCCS to be ignored.

`-o` *file*

Specifies the output file or directory. The output may only be a file if the input is a single file; otherwise, the output should be a directory into which the edited copies of the input files will be placed.

file...

Specifies the input files to be edited. If any of the specified files are directories, they will be recursively copied to the output directory and any HTML files within them updated.

RETURN CODE

The following table describes the return codes that the program displays when it exits.

TABLE 11 Exit Return Codes

Code	Description
0	The copy was successful.
1	A problem exists with the command-line arguments.
2	A problem exists with the command-line arguments.
3	An error occurred while performing the copy.

Detailed Example of `EditLinks` Command

In the following example, *test12-dir.wd* and *myworkdir.wd* represent file names that might exist on your system. Win32 users must also replace the UNIX system file separators with Windows file separators (/) to run these examples.

```
java -cp [jt-dir/lib/] javatest.jar com.sun.javatest.EditLinks -e
/work/ /java/jck-dev/scratch/12Jun00/jck-lab3/ -o test12_dir.wd
myworkdir.wd
```

13

Glossary

`.jtb` Files	See command file.
`.jte` Files	See environment files.
`.jti` Files	See configuration file.
`.jtp` Files	See parameter files.
`.jtr` File	See test result files.
`.jtx` Files	See exclude list.
`.kfl` Files	See Known Failures List File.

A

B

Batch Mode You can use either the `-batch` mode option or the current `-run` command to run tests from the command line or as part of a build process. Unless you are running tests from the command line and are using the GUI to monitor the test run, the `-batch` mode option is no longer required.

C

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 that represent the state of an object of that class and the behavior is given by a set of methods associated with the class.

Command File You can place routinely used configuration settings in a command file and include it in a command line. The command file is an ASCII file containing a lengthy series of commands and their arguments used in the command-line interface to modify specific configuration values before running tests.

You can use command files to configure and run tests, and write test reports either from the command line or as a part of a product build process. Using a command file allows you to repeatedly use a configuration without retyping the commands each time a test run is performed.

It is recommended that a descriptive name and the extension `.jtb` are used to help identify the function of each command file.

Configuration Information about the computing environment required to execute a test suite.

In the GUI, you can use the Configuration Editor to collect or modify configuration information or to load an existing configuration. See Configuration Editor. The Configuration Editor collects the following two types of data in a configuration file:

- Test environment
  Standard Values

In the command-line interface, you can perform the following tasks:

- Use the `EditJTI` utility to modify configuration information (see `EditJTI`).
- Set specific configuration values in the command line when starting the harness.

Configuration File Contains all of the information collected by the configuration editor about the test platform.

The harness derives the configuration values required to execute the test suite from environment entries in a configuration file (`.jti`).

Use the Configuration Editor or EditJTI to change configuration values in a `.jti` file.

You can also set specific values in the command line.

Configuration Value A value specified by the user for the purpose of configuring a test run.

Configuration values are derived from environment entries in a configuration file (`.jti`) and used by test suite specific plugin code to execute and run tests.

Use the Configuration Editor or `EditJTI` to change the configuration values in the `.jti` file. You can also set specific configuration values in the command line.

For legacy test suites the configuration value is read from an environment file (`.jte`). Current test suites do not use or support the environment file.

Current Configuration The configuration containing the test environment and standard values currently loaded in the test manager or specified in the command line for use in running tests and displaying test status.

D

E

EditJTI The JT harness provides an EditJTI utility that you can use from the command line to edit the values entered in a configuration file without opening the JavaTest Harness GUI. The EditJTI utility is the batch command equivalent of the JavaTest Harness Configuration Editor.

Environment See Test Environment.

Environment Entry A name-value pair derived from a configuration file and used by test suite specific plugin code to execute and run tests. These name-value pairs provide information (configuration values) about how to run tests of a test suite on a particular platform.

For legacy test suites, the name-value pairs are read from an environment file (`.jte`) and derived from the configuration file (`.jti`). Current test suites do not use or support the environment file.

Environment Files Contain one or more test environments used by legacy test suites. Environment files are identified by the `.jte` extension in the file name. Current test suites do not use or support the environment file.

Error	The test is not filtered out and the JT harness could not execute it. There are no test results for tests having errors. Errors usually occur because the test environment is not properly configured. In the GUI, the JT harness displays error icons for tests with errors and for folders containing any tests with errors. Folders marked with error icons can also contain tests and folders that are Failed, Not Run, Passed, and Filtered out.
Exclude List	Exclude list files (`*.jtx`), supply a list of invalid tests to be filtered out of a test run by the test harness. The exclude list provides a level playing field for all implementors by ensuring that when a test is determined to be invalid, then no implementation is required to pass it. Exclude lists are maintained by the technology specification Maintenance Lead and are made available to all technology licensees. In the GUI, use the configuration editor to add or remove exclude lists from a test run. In the command line, you can specify an exclude list in the command. To view the contents of an exclude list, choose Configure **->** Show Exclude List from the Test Manager menu bar. Exclude lists can only be edited or modified by the test suite Maintenance Lead.

F

Fail	Test results determined by the JT harness that do not meet passing criteria. In the GUI, the JT harness displays Failed icons for tests that the test suite has determined have failing results and for folders containing any tests with fail results. Folders marked with Failed icons can also contain tests and folders that are Not Run, Passed, and Filtered out.
Filtered Out	Folders and their tests that are excluded from the test run by one or more test run filters. In the GUI, Filtered Out folders and tests are identified in the test tree by grey folder and test icons.
Filters	A facility in the JT harness that accepts or rejects tests based on a set of criteria. There are two types of filters in the JT harness, view filters and run filters. View filters are set in the Test Manager to display the results for specific folders and tests and to create test reports. Run filters are set in the Configuration Editor or are specified as commands in the command-line to specify which tests are run.

G

H

HTTP Server Software included in the JT harness that services HTTP requests used to monitor a test run from a remote work station.

I

Interview File See configuration file.

J

JTI Standard file extension for a configuration file. See configuration file.

K

Keywords Special values in a test description that describe how the test is executed.

Keywords are provided by the test suite for use in the Configuration Editor or command line as a filter to exclude or include tests in a test run.

Known Failures List Files

Known Failures List (KFL) files (`*.kfl`), list tests that are known to fail.

The purpose of this option is to enrich the reporting output so you can monitor the status of certain tests across test runs. If a KFL is specified, the HTML report includes a Known Failure Analysis section. This section provides hypertext links to the tests listed in the KFL. Also, if bug IDs are specified in the KFL the IDs are hypertext linked to a bug tracking system (specified as a URL in File > Preferences > Test Manager > Reporting).

One or more KFLs can be specified as a parameter in the configuration editor. KFLs can also be specified from the command line.

L

M

N

O

Observer	An optional class instantiated from the command line to observe a test run. The class implements a specific observer interface.

P

Parameter Files	Legacy files used to configure how the harness runs legacy test suites on your system. Parameter files use the file name extension `.jtp`. Although parameter files are deprecated, the harness provides support for those test suites that use parameter files. Current test suites do not use or support the parameter files.
Pass	Test results determined by the JT harness to meet passing criteria. The JT harness displays Passed icons for tests that the test suite has determined have passing results and for folders containing only tests with passing results.
Port Number	A number assigned to the JT harness that is used to link specific incoming data to an appropriate service.
Prior Status	A filter used to restrict the set of tests in a test run based on the last test result information stored in the test result files (`.jtr`) in the work directory. Use the configuration editor or command line to enable the Prior Status filter for a test run.

Q

R

Report Directory	The directory in which the harness writes test reports. The location of the report directory is set in the GUI or from the command line by the user when generating test reports.

S

Standard Values	The Quick Set mode of the Configuration Editor displays the standard values of a configuration.
System Properties	Contains environment variables from your system that are required to run the tests of a test suite. Because the harness cannot directly access environment variables, you must use command-line options to copy them into the harness system properties.

T

Test Case	A subsection inside of a test. The terminology for a 'test case' varies among test harnesses, but in this harness, a test is a larger entity than the test case. A test contains zero or more test cases; a test with no test cases is simply a basic test entity in this harness. The 'test' is the primary entity rendered in the main tree display, summaries, etc. 'Test cases' are currently only recognized in limited areas of the harness.
Test Description	Machine readable information that describes a test to the JT 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. When using the JT harness, the test description is a set of test-suite-specific name-values pairs. Each test in a test suite has a corresponding test description that is typically contained in an HTML file.

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 legacy test suite is run, the harness gives the script a test environment containing environment entries from configuration data collected by the configuration editor. See configuration.

For legacy test suites, the environment entries were read from an environment file. Use of environment files is deprecated, but the harness continues to provide support for test suites that use environment files. See environment file.

Current test suites do not use or support environment files.

Test Manager The JT harness window used to configure, run, monitor, and manage tests from its panels, menus, and controls.

The Test Manager window is divided into two panes. It displays the folders and tests of a test suite in the tree pane on the left and provides information about the selected test or folder in the information panes on the right.

A new Test Manager window is used for each test suite that is opened.

Test Result Files Contains all of the information gathered by the JT harness during a test run.

The test result files (`.jtr`) are stored in a cache in the work directory associated with the test suite.

You can view the test result files in a web browser configured to use the JT harness `ResultBrowser` servlet.

Test Run Filters Include or exclude tests in a test run. Tests are included or excluded from test runs by the following means:

- Exclude lists
- Keywords
- Prior status

Test run filters are set using the Configuration Editor or the command-line interface.

Test Script A script used by the JT harness, responsible for running the tests and returning the status (pass, fail, error) to the harness. The test script must interpret the test description information returned to it by the test finder. The test script is a plug-in provided by the test suite. In the GUI, the Test Manager Properties dialog box lists the plug-ins that are provided by the test suite.

Test Suite A collection of tests, used in conjunction with the JT harness, to verify compliance of the licensee’s implementation of the technology specifications.

A test suite must be associated with a work directory before the harness can run its tests.

U

V

W

Work Directory A directory associated with a specific test suite and used by the JT harness to store files containing information about the test suite and its tests.

Until a test suite is associated with a work directory, the JT harness cannot run tests.

X

Y

Z

Index