SigTest
User's Guide
Version 3.0
E19036-02

November 2013
This guide describes how to install and run the SigTest collection of tools. This collection includes the Signature Test tool and the API Coverage tool. Signature Test tool includes utilities used to develop signature test components that can be used to compare API test signatures. API Coverage tool is used to estimate the test coverage a test suite provides for an implementation of a specified API.
# Contents

### Preface
- Documentation Accessibility .................................................................................................................. xiii
- Who Should Use This Guide .................................................................................................................... xiii
- Before You Read This Guide ................................................................................................................... xiii
- How This Guide Is Organized .................................................................................................................... xiv
- Related Documentation ............................................................................................................................ xiv
- Typographic Conventions ......................................................................................................................... xiv

### 1 Important Security Information

#### Introduction
- File Access Risks ........................................................................................................................................ 1-1
  - Path Vulnerabilities ................................................................................................................................. 1-1
  - File Creation Permissions ....................................................................................................................... 1-1
- Network Access Risks ............................................................................................................................... 1-2
- Remote Agent Risks ................................................................................................................................. 1-2

### 2 Introduction

#### Global Functionality
- Option File .................................................................................................................................................... 2-1
  - Option File Format ................................................................................................................................. 2-2
- The -version Option .................................................................................................................................. 2-2

### Part I Signature Test Tool

### 3 Introduction to Signature Test Tool

#### Signature Test Basics .......................................................................................................................... 3-1
#### What is Tested ....................................................................................................................................... 3-1
  - Mutual Binary Compatibility Check ........................................................................................................ 3-2
  - Mutual Source Compatibility Check ....................................................................................................... 3-2
  - Migration Binary and Source Code Check .............................................................................................. 3-3
  - Class and Class Member Attributes Checked .......................................................................................... 3-3
- Source and Binary Compatibility Modes ................................................................................................... 3-4
- Using Custom Signature Loaders .............................................................................................................. 3-5
4 Using the Signature Test Tool

Signature Test Tool Basics ................................................................. 4-1
  Reflection and Static Run Modes ................................................. 4-2
  Constant Checking in Differing Run Modes .................................. 4-2
  Generics Checking in Binary Mode ............................................. 4-2
  CLASSPATH and -classpath Settings ......................................... 4-3
    Bootstrap and Extension Classes ............................................. 4-3
Signature File Merge Rules .............................................................. 4-4
  Java ME Platform-based Merge .............................................. 4-4
Setup Command ......................................................................... 4-4
  Command Description .............................................................. 4-5
  Case Sensitivity of Option Arguments ....................................... 4-6
  Signature File Formats ............................................................. 4-7
  Signature File Contents ............................................................. 4-7
    Signature File Header ............................................................. 4-8
    Signature File Body ............................................................... 4-8
SignatureTest Command ............................................................... 4-9
  Command Description .............................................................. 4-10
Using Update Files .................................................................... 4-10
  Report Formats ........................................................................ 4-14
    Sorted Report ....................................................................... 4-15
    Unsorted Report .................................................................... 4-15
    Human-Readable Report ..................................................... 4-16
SetupAndTest Command ............................................................. 4-16
  Command Description .............................................................. 4-16
Merge Command ....................................................................... 4-18
  Command Description .............................................................. 4-18

Part II  API Coverage Tool

5 Introduction to API Coverage Tool

Static API Coverage Analysis .......................................................... 5-1
  Major Source of Error ............................................................... 5-1
  Advantages of Static Coverage Analysis .................................... 5-1
How It Works .............................................................................. 5-2
  Level of Accuracy During Analysis ......................................... 5-2
  Coverage Analysis Modes ....................................................... 5-3
  Filtering Coverage By Marking Up Signature Files .................. 5-3
    Filtering Markup Format .................................................... 5-4

6 Using the API Coverage Tool

Running API Coverage Tool .......................................................... 6-1
  Special Report File ................................................................. 6-3
  Exclude List ............................................................................ 6-3
Merging API Coverage XML Reports ........................................... 6-4
Part III API Check Tool

7 Introduction API Check Tool
The API Check Tool ........................................................................................................... 7-1

8 Using the API Check Tool
API Check Tool Verification Scope .................................................................................... 8-1
Running API Check Tool .................................................................................................... 8-2

Part IV Appendix

A Signature Test Tool Quick Start Examples
Example Programs ............................................................................................................. A-1
Example 1: Compare Two Different Implementations of the Same API ................................ A-2
Using the Setup and SignatureTest Commands ................................................................. A-2
Using the SetupAndTest Command .............................................................................. A-4
Example 2: Merge Two Signature Files ........................................................................... A-4
Running Merge Example .................................................................................................. A-5
Example Result Files ........................................................................................................ A-7

B API Coverage Tool Quick Start Examples
Example Procedures ......................................................................................................... B-1
Set Up the API Coverage Tool Ant Build Script ................................................................. B-1
Build API Coverage Tool Examples .................................................................................. B-2
Run the Example Test Suite ............................................................................................... B-2
Generate a Signature File for the Tested API ...................................................................... B-2
Use the API Coverage Tool to Calculate Test Coverage .................................................. B-3
Worst Case Mode .............................................................................................................. B-3
Real World Mode ............................................................................................................. B-4
Real World Mode Without Fields and Enum Constants, detail Level 5 ............................. B-5

C API Migration Compatibility Rules (Signature Test)
The Compatibility Rules ................................................................................................... C-1
General Rules .................................................................................................................... C-1
Interfaces and Annotation Types ....................................................................................... C-1
Interfaces and Class Methods ............................................................................................ C-2
Interfaces and Class Fields ............................................................................................... C-2
Classes ............................................................................................................................. C-3
Comments and Clarifications ............................................................................................ C-3
Rule 1.1 ............................................................................................................................. C-4
Rule 1.4 ............................................................................................................................. C-4
Rule 2.3 ............................................................................................................................. C-5
Rules 2.5, 2.6 ..................................................................................................................... C-5
Rule 3.6 ............................................................................................................................. C-6
Rule 4.2 ............................................................................................................................. C-6
D Merge Command Operative Principles

Element Handling by Merge

E Using the SigTest Tools With Ant

Using the Setup Command From an Ant Script
Using the SignatureTest Command From an Ant Script
Using the Merge Command From an Ant Script
Using APICheck From an Ant Script

F Using the SigTest Tools With Apache Maven

Goal Overview
Goal Descriptions
sigtest-plugin:setup
sigtest-plugin:test
sigtest-plugin:merge
apicover-plugin:apicover

G Custom SigTest Tool Example

Overview
Usage
Implementation
List of Figures

8-1  java.util Classes ........................................................................................................... 8-2
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-1</td>
<td>Settings for the Setup and SignatureTest Commands</td>
</tr>
<tr>
<td>4-2</td>
<td>Setup Command Options</td>
</tr>
<tr>
<td>4-3</td>
<td>Signature File Format Compatibility</td>
</tr>
<tr>
<td>4-4</td>
<td>Signature File Content Summary</td>
</tr>
<tr>
<td>4-5</td>
<td>SignatureTest Command Options</td>
</tr>
<tr>
<td>4-6</td>
<td>Update Types and Attributes</td>
</tr>
<tr>
<td>4-7</td>
<td>SetupAndTest Command Options</td>
</tr>
<tr>
<td>4-8</td>
<td>Merge Command Options</td>
</tr>
<tr>
<td>5-1</td>
<td>Example Scenarios and Potential Errors</td>
</tr>
<tr>
<td>6-1</td>
<td>API Coverage Tool Command Options</td>
</tr>
<tr>
<td>6-2</td>
<td>Report File Contents for Levels of Detail</td>
</tr>
<tr>
<td>6-3</td>
<td>Merge Command Options</td>
</tr>
<tr>
<td>C-1</td>
<td>API Migration Compatibility Rules - General</td>
</tr>
<tr>
<td>C-2</td>
<td>API Migration Compatibility Rules - Interfaces and Annotation Types</td>
</tr>
<tr>
<td>C-3</td>
<td>API Migration Compatibility Rules - Interfaces and Class Methods</td>
</tr>
<tr>
<td>C-4</td>
<td>API Migration Compatibility Rules - Interfaces and Class Fields</td>
</tr>
<tr>
<td>C-5</td>
<td>API Migration Compatibility Rules - General</td>
</tr>
<tr>
<td>E-1</td>
<td>Setup Attributes Available for Ant Scripts</td>
</tr>
<tr>
<td>E-2</td>
<td>SignatureTest Attributes Available for Ant Scripts</td>
</tr>
<tr>
<td>E-3</td>
<td>Required Setup Attributes Available for Ant Scripts</td>
</tr>
<tr>
<td>E-4</td>
<td>Optional Setup Attributes Available for Ant Scripts</td>
</tr>
<tr>
<td>E-5</td>
<td>SAMPLE of REDOING TABLE UNITS: Setup Attributes Available for Ant Scripts</td>
</tr>
<tr>
<td>F-1</td>
<td>The sigtest-plugin Goals</td>
</tr>
<tr>
<td>F-2</td>
<td>Required Parameters (sigtest-plugin:setup)</td>
</tr>
<tr>
<td>F-3</td>
<td>Optional Parameters (sigtest-plugin:setup)</td>
</tr>
<tr>
<td>F-4</td>
<td>Required Parameters (sigtest-plugin:test)</td>
</tr>
<tr>
<td>F-5</td>
<td>Optional Parameters (sigtest-plugin:test)</td>
</tr>
<tr>
<td>F-6</td>
<td>Required Parameters (sigtest-plugin:merge)</td>
</tr>
<tr>
<td>F-7</td>
<td>Optional Parameters (sigtest-plugin:merge)</td>
</tr>
<tr>
<td>F-8</td>
<td>Required Parameters (apicover-plugin:apicover)</td>
</tr>
<tr>
<td>F-9</td>
<td>Optional Parameters (apicover-plugin:apicover)</td>
</tr>
</tbody>
</table>
## List of Examples

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>Option File ................................................................. 2-2</td>
</tr>
<tr>
<td>4-1</td>
<td>Update File Example ...................................................... 4-12</td>
</tr>
<tr>
<td>4-2</td>
<td>Unsorted Report Example ................................................. 4-15</td>
</tr>
<tr>
<td>5-1</td>
<td>Signature File Mark-Up.................................................... 5-3</td>
</tr>
<tr>
<td>A-1</td>
<td>Version 1.0 of test.java.................................................. A-1</td>
</tr>
<tr>
<td>A-2</td>
<td>Version 2.0 of test.java.................................................. A-1</td>
</tr>
<tr>
<td>A-3</td>
<td>Contents of ./x1.sig......................................................... A-7</td>
</tr>
<tr>
<td>A-4</td>
<td>Contents of ./x2.sig......................................................... A-7</td>
</tr>
<tr>
<td>A-5</td>
<td>Contents of ./x3.sig......................................................... A-7</td>
</tr>
<tr>
<td>A-6</td>
<td>Contents of ./x1+x2.sig.................................................... A-8</td>
</tr>
<tr>
<td>F-1</td>
<td>Sample Plugin Configuration (pom.xml)............................. F-2</td>
</tr>
<tr>
<td>F-2</td>
<td>Sample Plugin Configuration (pom.xml)............................. F-4</td>
</tr>
<tr>
<td>F-3</td>
<td>Sample Plugin Configuration (pom.xml)............................. F-5</td>
</tr>
<tr>
<td>F-4</td>
<td>Sample Plugin Configuration (pom.xml)............................. F-7</td>
</tr>
</tbody>
</table>
This guide describes how to install and run the SigTest collection of tools. This collection includes the Signature Test tool and the API Coverage tool. Signature Test tool includes utilities used to develop signature test components that can be used to compare API test signatures. API Coverage tool is used to estimate the test coverage a test suite provides for an implementation of a specified API.

Note: For simplicity, this user’s guide refers to the test harness as the JavaTest harness. Note that the open source version of the harness, called JT harness, can be used in its place. The JT harness software can be downloaded from: http://jtharness.java.net/

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/us/corporate/accessibility/index.html.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/support/contact.html or visit http://www.oracle.com/accessibility/support.html if you are hearing impaired.

Who Should Use This Guide

This guide is for developers of quality assurance test suites and developers of compatibility test suites — TCKs for a Java platform API as part of the Java Community Process (JCP) program.

Before You Read This Guide

Before reading this guide, it is best to be familiar with the Java programming language. A good resource for the Java programming language is the Oracle Technical Network web site, located at http://www.oracle.com/technetwork/java/index.html.

Note: Web URLs provided are subject to change.
How This Guide Is Organized

Introduction describes the SigTest collection of tools.

Part I describes how you can use the Signature Test tool to easily compare the signatures of two different implementations of the same API.

Part II describes how you can use the API Coverage tool to estimate the test coverage a test suite provides for an implementation of a specified API.

Part III describes how you can use the API Check tool to track API changes and roughly check for source and binary compatibility.

Part IV contains an appendix that includes step-by-step examples that show how to use Signature Test tool.

Related Documentation

For details about the Java programming language, see the specifications at http://docs.oracle.com/javase/specs/ and the many Java programming texts.

Typographic Conventions

<table>
<thead>
<tr>
<th>Typeface</th>
<th>Meaning</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>AaBbCc123</td>
<td>The names of commands, files, and directories, or on-screen computer output</td>
<td>Edit your .login file.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Use ls -a to list all files.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>% You have mail.</td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>Book titles, new words or terms, words to be emphasized</td>
<td>Read Chapter 6 in the User’s Guide.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>These are called class options.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>You must be superuser to do this.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>To delete a file, type rm filename.</td>
</tr>
<tr>
<td>'''' or '^'</td>
<td>A backslash at the end of a line indicates that a long code line has been broken in two on a UNIX® system, typically to improve legibility in code. The caret character (^) indicates this on a Microsoft Windows system.</td>
<td>java classname \</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[classname_arguments]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>java classname ^</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[classname_arguments]</td>
</tr>
<tr>
<td>Indented code or command line</td>
<td>Indicates a wrapped continuation from a previous line with no carriage return or return character in the actual code.</td>
<td>java classname</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[classname_arguments]</td>
</tr>
</tbody>
</table>

1 The top-most SigTest Tool collection installation directory is referred to as SigTest-Directory throughout the SigTest Tool collection documentation.
This section describes known SigTest product vulnerabilities that can be exploited by authorized or unauthorized users or others who have access to the network where the SigTest product runs.

This chapter contains these topics:

- Section 1.1, "Introduction"
- Section 1.2, "File Access Risks"
- Section 1.3, "Network Access Risks"
- Section 1.4, "Remote Agent Risks"

1.1 Introduction

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

1.2 File Access Risks

The SigTest product tools read, write, and deletes files and can therefore potentially expose or damage information stored in files.

1.2.1 Path Vulnerabilities

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

1.2.2 File Creation Permissions

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

1.3 Network Access Risks

The SigTest tools are intended to be run in a semi-trusted environment. Never expose the tools directly to the Internet, which can provide a path for malicious intrusion. Most test scenarios require no network and can be run on a standalone machine. If you must run tests on a networked machine, the most secure configuration is a local network that is physically isolated from organizational intranets and sensitive information.

1.4 Remote Agent Risks

The SigTest tools permit the use of custom remote signature loaders (see "Using Custom Signature Loaders" on page 5). Such a remote signature loader uses open communication ports on the SigTest host computer and on a remote computer; therefore you must ensure that both machines are protected from malicious attack. You must also ensure that the remote class loader itself is not malicious. For the most secure operation, connect the SigTest host and remote computers only to a protected intranet on a physically isolated network.
The SigTest product is a collection of tools that can be used to compare APIs and to measure the test coverage of an API. The tools were originally created to assist in the creation of Java technology compatibility test suites (TCKs), but are also useful in the creation of other types of test suites and in the software development process. The SigTest product consists of the following tools.

**Signature Test tool** makes it easy to compare the signatures of two different implementations or different versions of the same API. When it compares different implementations of the same API, the tool verifies that all of the members are present, reports when new members are added, and checks the specified behavior of each API member. When it compares different versions of the same API, the tool checks that the old version can be replaced by the new one without adversely affecting existing clients of the API.

**API Coverage tool** can be used to estimate the test coverage a test suite provides for an implementation of a specified API. It does this by determining how many public class members the test suite references within the API specification. The tool uses a signature file representation of the API specification as the source of specification analysis. It does not process a formal specification in any form.

**API Check tool** tracks API changes and roughly checks for source and binary compatibility. Much like Signature Test tool, it compares the reference implementation of an API recorded as golden signature file with a tested implementation. Unlike Signature Test tool, which requires that all classes (and the classes they depend on) be specified to the tool, API Check tool does not require dependencies to be specified. For that reason, API Check tool is faster than Signature Test tool, but less rigorous.

Part I of this manual describes how to use the Signature Test Tool, Part II describes how to use the API Coverage tool, Part III describes how to use the API Check tool, and Part IV contains two appendices that include step-by-step examples that show how to use the Signature Test tool and the rules used in the Signature Test API migration feature.

## 2.1 Global Functionality

This section describes functionality that is available to all the tools and their commands.

### 2.1.1 Option File

All of the tools accept options from an option file as well as from the command line. Option files allow you to create complex command lines in a text editor and then reuse them.
The option file is specified on the tool command line using the “@” character as shown in the following example.

```
java -jar sigtest.jar test @file
```

In this case, `file` is the path to a option file. You can use a combination of option files and command-line options as shown in the following example.

```
java -jar sigtest.jar test @file -package com.acme.api.printing.color
```

### 2.1.1.1 Option File Format

In addition to options and their arguments, option files can also contain “set” commands that you can use to set variables for use within the option file. The option file syntax is demonstrated in Example 2–1.

**Example 2–1 Option File**

```
#--------------------------------------------------
#This option file checks the API against JRE 1.6
#--------------------------------------------------

set HOME=/home/ersh
set JRE=/opt/java/jre16
set AMCE_HOME=$(HOME)/projects/release
set API=$(AMCE_HOME)/lib/ams.jar

static
backward
classpath=$(JRE)/lib/rt.jar:$(API)
filename=$(AMCE_HOME)/misc/reference.sig
```

This option file is equivalent to the following command line.

```
java -jar sigtest.jar test -static -backward -classpath /opt/java/jre16/lib/rt.jar:/home/ersh/projects/release/lib/ams.jar
-filename /home/ersh/projects/acmesystem/msc/reference.sig
-package com.acme.api.printing.color
```

### 2.1.2 The -version Option

All of the tools and commands support the `-version` option as a way to display information about the tools and will list which signature files they are compatible with. In addition, you can specify the `-version` option as an argument directly to the JAR file as well as shown here.

```
java -jar sigtest.jar -version
```
Part I
Signature Test Tool

This part describes how you can use the Signature Test Tool to easily compare the signatures of two different implementations or different versions of the same API.

This part includes the following chapters:

- Chapter 3, "Introduction to Signature Test Tool"
- Chapter 4, "Using the Signature Test Tool"
You can use the Signature Test tool to easily compare the signatures of two different implementations of the same API. It verifies that all of the members are present, reports when new members are added, and checks the specified behavior of each API member.

### 3.1 Signature Test Basics

A signature test compares two implementations of an API and reports the differences. The Signature Test tool compares the signatures of two implementations or versions of the same API and can do the following:

- Create and run a test that verifies that all of the members are present
- Report when new members are added
- Check the specified behavior of each API member
- Determine whether the old version of the API can be replaced with the newer one without adversely affecting existing clients of the API

The signature test created by the Signature Test tool can be run independently at the command line, or under the control of the JavaTest™ harness.

---

**Note:** For simplicity, this user’s guide refers to the test harness as the JavaTest harness. Note that the open source version of the harness, called JT harness, can be used in its place. The JT harness software can be downloaded from this URL: [http://jtharness.java.net/](http://jtharness.java.net/)

---

The Signature Test tool was originally created to assist in the creation of Java technology compatibility test suites (TCKs). It simplified the process of verifying that the API signature of a new implementation of a Java technology matched the signature of a reference implementation of that API.

When used in a software development environment, the Signature Test tool can be used to track and control changes to an API throughout the development process.

### 3.2 What is Tested

The signature test algorithm compares the API implementation under test with a signature file created from the API you are comparing it to — often referred to as a reference implementation. The signature test checks for compatibility by verifying the equality of API member sets. By checking for mutual compatibility of API member sets, the test verifies that the following conditions are true:
If an API item is defined in the reference implementation of the API, then that item is implemented in the API under test, and vice versa.

Attributes chosen for comparison are identical in both implementations of the API. The tool chooses attributes for comparison according to the type of check being processed. This is described more in "Mutual Binary Compatibility Check" on page 3-2 and "Mutual Source Compatibility Check" on page 3-2.

By checking for migration compatibility of API member sets, the test verifies that the following conditions are true:

- If an API item is defined in the previous version of the API, then that item is implemented in the API version under test.
- If an API item is not defined in the previous version of the API, but added to the API version under test, the added item does not break backward compatibility. This is described more in "Migration Binary and Source Code Check" on page 3-3.
- Attributes chosen for comparison are identical in both versions of the API or their change does not break backward compatibility. The tool chooses attributes for comparison according to the type of check being processed. This is described more in "Migration Binary and Source Code Check" on page 3-3.

### 3.2.1 Mutual Binary Compatibility Check

The signature test binary compatibility check mode verifies that a Java technology implementation undergoing compatibility testing and its referenced APIs are mutually binary compatible as defined in Chapter 13, “Binary Compatibility,” of The Java Language Specification. This assures that any application runs with any compatible API without any linkage errors.

This check is less strict than the default source compatibility check, described next. It is for use primarily in the special case of when a technology is developed for Java technology environments that are purely runtime. Such an environment does not provide a Java technology-based compiler (Java compiler), nor does it include class files that could be used to compile applications for that environment. Because of the limited use of such an environment, the API requirements are slightly relaxed compared to environments that support application development.

Java application environments can contain several Java technologies. Not all Java technologies can be combined with each other, and in particular, their sets of API signatures might be incompatible with each other. Relaxing signature checks to the level of mutual binary compatibility allows the developer to combine technologies in a purely runtime environment that cannot be combined otherwise.

### 3.2.2 Mutual Source Compatibility Check

While binary compatibility is important, it cannot guarantee that an application in binary form as a set of class files can be recompiled without error.

The signature test source compatibility check mode verifies that any application that compiles without error with a compatible API, compiles without error with all other source compatible APIs.

Mutual source compatibility is a stricter check than the mutual binary compatibility and the Signature Test tool performs it by default.
3.2.3 Migration Binary and Source Code Check

Mutual compatibility is generally used in certification processes where the goal is to ensure that an alternative or third-party implementation of an API conforms to a reference implementation. Application developers have a different concern, they must ensure that evolving library APIs that their applications link to continue to work with customers’ applications. The Signature Test tool can be used to check APIs as they evolve and ensure both binary and source code migration compatibility.

Migration binary compatibility checking ensures that there will be no linkage errors between pre-existing client binaries and the new version. This determination is based on the Chapter 13, “Binary Compatibility,” of The Java Language Specification.

Migration source code compatibility means that pre-existing client source code can be recompiled with the new version without compilation errors.

3.2.4 Class and Class Member Attributes Checked

A Java platform API consists of classes, and interfaces, and their member fields, methods, and constructors, and documented annotations. In turn, all of these API items can have various attributes such as names, modifiers, a list of parameters, a list of interfaces, exceptions, nested classes, and so forth. A signature test checks that certain members and attributes belonging to the API under test are the same as those defined by the API to which it is being compared. Signature test only checks public and protected API items and in the most cases ignores private and package access items. Members with private and package visibility are taken into account when the tool calculates hiding for more visible members.

The tool checks the following attributes when comparing items in the API implementation under test:

- **Classes and interfaces, including nested classes and interfaces:**
  - Set of modifiers except `strictfp`
  - Name of the superclass
  - Names of all superinterfaces, direct plus indirect, where order is insignificant

- **Constructors:**
  - Set of modifiers
  - List of argument types
  - In source compatibility mode only, the normalized list of thrown exceptions, where order is insignificant

  Normalizing the throw lists involves removing all superfluous exception classes. An exception class is superfluous if it is a subclass of either the `java.lang.RuntimeException` class, the `java.lang.Error` class, or another class from the same list.

- **Methods:**
  - The set of modifiers, except `strictfp`, `synchronized`, and `native`
  - The return type
  - The list of argument types
  - In source mode only, the normalized list of thrown exceptions, described earlier, where order is insignificant

- **Fields:**
- Set of modifiers, except transient and volatile
- Field type

- Documented annotations with SOURCE and RUNTIME retention of the following types:
  - Classes and interfaces
  - Fields, methods and constructors
  - Parameters and annotation types

The tool performs the check in the following order:

1. For all top-level public and protected classes and interfaces, it compares the attributes of any classes and interfaces with the same fully qualified name.

2. Taking into account all declared and inherited members, it compares all public and protected members of the same kind and same simple name, treating constructors as class members for convenience sake.

### 3.3 Source and Binary Compatibility Modes

Earlier Signature Test tool versions performed a comparison of all exceptions declared in throws clauses for methods and constructors. Certain variations in this area caused an error message during the signature test. Despite these error messages, the source files compiled successfully together. Successful compilation is the basic criteria for source compatibility with the current Signature Test tool, while successful linking as the basic criteria for binary compatibility.

Changes to the throws clause of methods or constructors do not break compatibility with existing binaries because these clauses are checked only at compile time, causing no linkage error. For the purpose of signature testing, this relates directly to binary compatibility as described earlier in "Mutual Binary Compatibility Check" on page 3-2.

The adaptation of JSR 68, The Java ME Platform Specification, formalized the use of building blocks in API development. A building block is a subset of an existing API that is approved for reuse in the construction of profiles or optional packages. The building block concept enables a developer to duplicate the functionality provided by another API without having to redefine an entirely new API. For further details see JSR 68 at [http://www.jcp.org/en/jsr/detail?id=68](http://www.jcp.org/en/jsr/detail?id=68).

The use of building blocks created a need for more lenient checking of exception throw lists compared to earlier Signature Test tool versions. Consequently, Signature Test tool 2.2 provides both a source and a binary compatibility mode of operation. This retains compatibility with earlier signature files while adding support for building blocks and eliminating the unnecessary error messages.

The SignatureTest command recognizes the -mode option that takes the values “src” or “bin” as arguments for choosing source mode or binary mode. The choice of which mode to use depends on the type of signature file being used in the test. This is described in more detail later in these sections:

- "Setup Command" on page 4-4 describes how to generate a signature file
- "SignatureTest Command" on page 4-9 describes how to specify the mode when running a signature test
- "Merge Command" on page 4-18 describes how to generate a combined signature file from set of signature files
The difference between the binary and source compatibility modes is how the tool handles the throws list for constructors and methods (as described in "Class and Class Member Attributes Checked" on page 3-3). Constant checking behavior is also different in binary and source compatibility modes. Although constant checking can be applied to binary compatibility, it is a necessary prerequisite for source code compatibility. "Constant Checking in Differing Run Modes" on page 4-2 describes these differences in more detail.

3.4 Using Custom Signature Loaders

The signature test has a requirement for the Java Platform, Standard Edition (Java SE platform) runtime environment version 1.4 or later. This requirement might prevent use of the tool on limited or nonstandard environments such as some Java Platform, Micro Edition (Java ME platform) or Java Platform, Enterprise Edition (Java EE platform) configurations.

To overcome this, the tool provides support for custom signature loaders that can be implemented as plug-ins. These plug-ins gather signatures from a runtime environment when the SignatureTest command cannot be run directly. For example, you might create a light-weight remote JavaTest harness agent and run the signature loader on a remote Connected Device Configuration (CDC) compatible device. Another example is using a wrapped J2EE platform bean as a signature loader inside a J2EE platform container where any direct file I/O operations are prohibited.

As an aid in developing such an extension, the Signature Test tool distribution includes a class library that contains a signature serializer and some related utility classes in the SigTest-Directory/lib/remote.jar file. This file contains a subset of the Signature Test tool classes that are necessary to develop a custom plug-in. All of these library classes are CDC 1.0 compatible and have minimal memory requirements. The source code for these classes is distributed in the SigTest-Directory/redistributables/sigtest_src.zip file. The code is designed for running a plug-in with the JavaTest harness using the Java ME Framework. The server and client source code and the HTML test descriptions for an actual plug-in example are located in the SigTest-Directory/examples/remote directory.

Note: The open source version of the ME Framework is available at: http://cqme.java.net/framework.html.
Using the Signature Test Tool

This chapter provides a synopsis of each of the Signature Test tool commands along with their available options and arguments. It contains these sections:

- **Signature Test Tool Basics**
- **Setup Command**
- **SignatureTest Command**
- **SetupAndTest Command**
- **Merge Command**
- **Report Formats**

**Note:** Appendix A includes examples of each command.

### 4.1 Signature Test Tool Basics

The Signature Test tool operates from the command line to generate or manipulate signature files. A signature file is a text representation of the set of public and protected features provided by an API. Test suite developers include it in a finished test suite as a signature reference for comparison to the technology implementation under test. The following list shows the commands that are available.

- **Setup** - Creates a signature file from either an API defined by a specification or a reference API implementation.
- **SignatureTest** - Compares the reference API represented in the signature file to the API under test and produces a report. This is the test that becomes part of a finished test suite.
- **SetupAndTest** - Executes the **Setup** and **SignatureTest** commands in one operation.
- **Merge** - Creates a combined signature file from several signature files representing different Java APIs in one Java runtime environment according to the JSR 68 rules.

The Signature Test tool distribution includes a Java Archive (JAR) file used for developing a signature test and one for distribution within a finished test suite to run its signature test. The description of each follows:

- **sigtestdev.jar** - Contains classes for running the commands used during signature test development.
- **sigtest.jar** - Contains only the classes for running the **SignatureTest** command. This file is distributed in a finished test suite.
Test suite developers perform these operations while using `sigtestdev.jar` to develop a signature test.

1. Run the `Setup` command to create a signature file from either an API defined by a specification or a reference API implementation.
2. Include the files required to run the signature test in the finished test suite distribution.

### 4.1.1 Reflection and Static Run Modes

Two run modes are available during command execution. These modes determine how the class descriptions are examined and retrieved, as follows:

- **Reflection Mode** - Uses reflection to examine API classes and retrieve information about them. The reflection mode is of greatest advantage when the API to be analyzed has no external class files.
- **Static Mode** - Specified with the `-static` flag, the tool parses only the class files listed in the `-classpath` command-line option.

**Note:** In static mode you can test specified classes in another runtime environment. For example, this can be useful to analyze APIs that are part of a Java SE platform 1.4.2 environment when the `SignatureTest` command is run on a Java SE platform version 5.0.

### 4.1.2 Constant Checking in Differing Run Modes

The requirements related to constant checking differ in binary and source compatibility testing. Although constant checking can be applied to binary compatibility, it is a necessary prerequisite for source code compatibility. Use the `-static` mode to enforce strict constant checking in source code compatibility testing.

When running a signature test in source compatibility mode and using the static mode, constant checking is strict and two way. This means that all the constant fields specified in the reference API must exist and have the same values in the API under test. Likewise, all the constant fields and their related values specified in the API under test must exist and have the same values in the reference API.

In binary compatibility mode, the requirements related to constant checking are less strict. The signature test verifies that all the constant fields and associated values contained in the reference API are also available in the API under test. If any field values are missing or different, it reports an error. However, the signature test does not report an error if constant values are found in the API under test that are not available in the reference API.

### 4.1.3 Generics Checking in Binary Mode

The information related to generics is not used by the Java Virtual Machine\(^1\) at runtime. This information is used only by the compiler at compile time. Because the information does not effect the runtime linkage process, it cannot cause binary incompatibility.

In binary mode the `SignatureTest` command compares the signatures of parameterized types after omitting the type parameters and arguments from both the signature file and the analyzed API (termed “type erasure”). This is to ensure that they

\(^1\) The terms “Java Virtual Machine” and “JVM” mean a Virtual Machine for the Java platform.

The bridge methods that are generated by the compiler during type erasure are not a part of the API and so they are ignored by the Signature Test tool.

### 4.1.4 CLASSPATH and -classpath Settings

Table 4–1 lists the requirements for setting the CLASSPATH environment variable and the -classpath option when running either the Setup or SignatureTest commands. The table uses the following terms to describe the classes that must be included:

- **Required classes** - All superclasses and superinterfaces of the classes under test
- **Classes under test** - The set of classes specified by a combination of the following options:
  - -package
  - PackageWithoutSubpackages
  - -exclude

**Note:** The Setup command can only be run in static mode.

<table>
<thead>
<tr>
<th>Variable or Option</th>
<th>In Reflection Mode</th>
<th>In Static Mode</th>
</tr>
</thead>
</table>
| CLASSPATH environment variable | Required. Must contain the following:
  - sigtestdev.jar or sigtest.jar  
  - Classes under test
  - Other required classes, except for bootstrap and extension classes described in 'Bootstrap and Extension Classes' on page 4-3 | Required. Must contain either sigtestdev.jar or sigtest.jar. |
| -classpath argument | N/A | Required. Must contain the following:
  - Classes under test
  - All required classes |

### 4.1.4.1 Bootstrap and Extension Classes

Bootstrap and extension classes are those located in rt.jar and several other JAR files under the Java-Home/jre/lib/ directory, where Java-Home is the base directory of the Java platform runtime installation. For example, classes from the java package such as java.lang.Object are bootstrap classes. Their location is already available to the Java Virtual Machine environment. Because of this, they do not need to be specified in the CLASSPATH environment variable when reflection mode is used. Furthermore, bootstrap and extension classes are always loaded from JAR files located in the Java-Home/jre/lib/ directory, regardless of whether they were specified in the class path or not. This is an important feature of the reflection mode.
4.2 Signature File Merge Rules

The -Files option of the SignatureTest command accepts values to specify one or more signature files that are combined to represent an API configuration that is used as input for testing. This resulting API-set can also be combined into a single signature file for testing purposes.

By default the API combination is assumed to be constructed according to the JSR 68 rules. This can be overridden during a Signature Test tool test run by specifying the -NoMerge option to run the signature test. The -NoMerge option forces the Signature Test tool to use the first class description it encounters if more than one class with the same name is found in the set of signature files specified by the -Files option. In this case it uses the class from the left-most signature file that is specified with -Files.

Note: The operative merging principles used by the Signature Test tool are described in Appendix D.

4.2.1 Java ME Platform-based Merge

The Java ME platform architecture specified by JSR 68\(^1\) allows for the inclusion of several Java platform APIs in one conforming Java platform runtime environment. The condition for combining these APIs is that any application written for the resulting runtime environment must execute successfully on the combination. See the JSR 68 specification for details.

If such a combination exists, it is defined on the basis of the combined sets of APIs with semantics, and the semantics must be compatible with those of all the included components.

The only means of verifying the semantics of the combination is to run the applicable test suite for each API. However, it is possible to automate the creation of a combined set of API signatures, and it is also possible to detect when a combined set cannot be built.

The Merge command functionality combines (merges) several input signature files into one resulting signature file, as follows: If A, B, and C are signature files, then A + B yields signature file C, and each of the three signature files represent the corresponding classes of their respective APIs. The Merge process constructs the resulting API C out of the two input APIs A and B.

4.3 Setup Command

The Setup command has the following synopsis:

```
java -jar sigtestdev.jar Setup [options]
```

or

```
java com.sun.tdk.signaturetest.Setup [options]
```

Table 4–2 describes the available command options and the values that they accept.

Before running the command, also see these sections: “Case Sensitivity of Option Arguments” on page 4-6 and "CLASSPATH and -classpath Settings" on page 4-3.

\(^1\) http://jcp.org/en/jsr/detail?id=68
4.3.1 Command Description

The Setup command accepts a reference implementation of an API as input. The command processes the API input to generate a signature file that represents the API to be used as a reference of comparison for the purpose of signature testing.

Setup processes the API input in the static mode by parsing the set of classes specified with the -classpath options.

Also see Appendix A for an example of the command that you can run.

Note: Arguments to options marked with an asterisk (*) in Table 4–2 can be specified multiple times. For example, if you want to specify filtering for the java.awt, java.lang, and java.math packages, you can specify the three arguments separated by a file separator character as shown in the following example: -Package java.awt:java.lang:java.math On Unix operating systems the separator character is “;” and on Microsoft Windows it is “;”.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-help</td>
<td>Optional. Displays usage information for available command options and exits.</td>
</tr>
<tr>
<td>-debug</td>
<td>Optional. Enables printing of the stack trace for debugging purposes if Setup fails.</td>
</tr>
<tr>
<td>-classpath path</td>
<td>Required. Specifies the path to one or more APIs that generate the signature file. Can contain multiple directories or ZIP or JAR files. The -package option further refines the set of classes specified in -classpath (see &quot;CLASSPATH and -classpath Settings&quot; on page 4-3). There is no default -classpath. Use the path separator appropriate for the platform (identified by java.io.File.pathSeparator).</td>
</tr>
<tr>
<td>-TestURL path</td>
<td>Optional. Specifies the directory location in which to create the signature file as a file protocol URL: file://path Must end in a trailing slash on a UNIX system or a backslash on a Microsoft Windows or DOS system. Setup does not support the HTTP protocol.</td>
</tr>
<tr>
<td>-FileName file_name</td>
<td>Required. Specifies the name of the signature file to be created.</td>
</tr>
<tr>
<td>-ClosedFile</td>
<td>Optional. The default if not specified. Specifies to include in the signature file all direct and indirect superclasses for all required classes (tested classes), even if these superclasses are non-public or from untested packages.</td>
</tr>
<tr>
<td>-NonClosedFile</td>
<td>Optional. Declines the default -ClosedFile mode previously described. Does not include all direct and indirect superclasses and superinterfaces of tested classes in the signature file</td>
</tr>
</tbody>
</table>
### 4.3.2 Case Sensitivity of Option Arguments

The specification of each argument option at the command line is not case sensitive, but the input value entered immediately after the option is case sensitive.

The following two command lines produce identical results for the `-FileName` flag:

```plaintext
% java -jar sigtestdev.jar Setup -FileName name.sig
% java -jar sigtestdev.jar Setup -filename name.sig
```

However, these two might not produce identical results if the host operating system is case sensitive to the file name values entered:

```plaintext
% java -jar sigtestdev.jar Setup -FileName name.sig
% java -jar sigtestdev.jar Setup -FileName NAME.sig
```
4.3.3 Signature File Formats

The Signature Test tool has changed signature file formats through progressive versions. Table 4–3 lists the existing signature file formats and describes how each relates to a specific Signature Test tool version. In Signature Test tool 2.2, the SignatureTest and Merge commands read v2.1 and later signature files, and output only v4.1.

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0</td>
<td>Generates a signature file with simple class member names. This was the default format in Signature Test tool 1.0, but is not supported by SignatureTest command in Signature Test tool 1.3 and later.</td>
</tr>
<tr>
<td>v1</td>
<td>Generates a signature file with fully qualified class member names. This was the default format in Signature Test tool 1.1. This format includes non-normalized exception throw lists for constructors and methods. Normalizing the throw list involves removing all superfluous exception classes. A class is superfluous if it is a subclass of either the java.lang.RuntimeException class, or the java.lang.Error class, or another class from the same list. This format is not supported by SignatureTest command in Signature Test tool 1.3 and later.</td>
</tr>
<tr>
<td>v2</td>
<td>This is the default format for Signature Test tool 1.2. Generates a signature file with fully qualified class member names and modified supr statements. This format includes normalized exception throw lists for constructors and methods. This format is not supported by SignatureTest command in Signature Test tool 1.3 and later.</td>
</tr>
<tr>
<td>v2.1</td>
<td>This version extends the v2 format to indicate whether an interface is inherited directly or indirectly. It is read by SignatureTest command in Signature Test tool 1.3 and later.</td>
</tr>
<tr>
<td>v3.1</td>
<td>Generates data for JDK™ software version 5.0 such as generics, annotations, and enums.</td>
</tr>
<tr>
<td>v4.0</td>
<td>Inherited members are not written to the signature file. Private and default visibility fields and nested classes that can potentially hide visible API elements are tracked. In Signature Test tool 2.2, all output files are of this version.</td>
</tr>
<tr>
<td>v4.1</td>
<td>Non-static constants are written to the signature file. References to outer classes are stored for the inner classes. Constructor names are changed to &lt;init&gt;. This has been the case since SigTest 2.2.</td>
</tr>
<tr>
<td>v4.2</td>
<td>Support for Java 8 features, such as default and static methods in interfaces, and type and repeating annotations. The Setup command writes files in this format since Signature Test tool 3.0.</td>
</tr>
</tbody>
</table>

Also see "Source and Binary Compatibility Modes" on page 3-4.

4.3.4 Signature File Contents

Setup generates each signature file with a mandatory header in the first two lines, followed by the body of the signature file.

---

**Note:** Comment lines start with the pound (#) character and can be inserted anywhere after the first two mandatory header lines.
4.3.4.1 Signature File Header
Setup generates the first two mandatory header lines of each signature file as follows:

#Signature file format
#Version version-string

With the following variable replacement values:

- *format* is either one of the values described in Table 4–3, or empty, that indicates v0.
- *version-string* is a value taken directly from the argument given at the startup command line to the -apiVersion option (see Table 4–2).

Note that the header can also contain an optional copyright string.

4.3.4.2 Signature File Body
The remaining body of a signature file immediately follows the header. It contains the following information, which is further clarified in Table 4–4:

- For each public or protected class:
  - All modifiers except strictfp
  - The fully qualified name of any superclass or interfaces implemented
  - Generic type parameters, and annotations.

- For each public or protected interface:
  - All modifiers except strictfp
  - The fully qualified name of any superinterfaces implemented,
  - Generic type parameters
  - Annotations

- For each public or protected field:
  - All modifiers except transient and volatile
  - The fully qualified name of the field’s type and its fully qualified name
  - If the field is a primitive or string constant, the value of the field is included
  - Documented annotations and type parameters

- For each public or protected method:
  - All modifiers (except native, synchronized, and strictfp)
  - The fully qualified name of the type of returned value, the method’s fully qualified name
  - Types of all parameters
  - The names of exceptions declared in a throws clause

- For each public or protected constructor:
  - All modifiers
  - The fully qualified name of the constructor
  - Types of all parameters
  - Any exceptions declared in a throws clause
Table 4–4 further summarizes the contents of a generated signature file. A plus (+) indicates a class modifier is included in a generated signature file and a minus (-) indicates it is ignored for that particular element. A blank cell indicates that the condition does not apply to a cell, for example, a class does not have a transient modifier so it is blank.

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Class or Interface (Including Nested)</th>
<th>Field</th>
<th>Method</th>
<th>Constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>++</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>protected</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>abstract</td>
<td>+</td>
<td></td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>static</td>
<td>+</td>
<td>+</td>
<td></td>
<td>+</td>
</tr>
<tr>
<td>final</td>
<td>+</td>
<td></td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>strictfp</td>
<td></td>
<td></td>
<td></td>
<td>–</td>
</tr>
<tr>
<td>transient</td>
<td></td>
<td></td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>volatile</td>
<td></td>
<td></td>
<td>–</td>
<td></td>
</tr>
<tr>
<td>synchronized</td>
<td></td>
<td></td>
<td>–</td>
<td></td>
</tr>
<tr>
<td>native</td>
<td></td>
<td></td>
<td>–</td>
<td></td>
</tr>
<tr>
<td>varargs</td>
<td></td>
<td></td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

4.4 SignatureTest Command

The SignatureTest command has the following synopsis:

```java
java -jar sigtest.jar Test [options]
```

or

```java
java com.sun.tdk.signaturetest.SignatureTest [options]
```

The second alternative assumes that sigtest.jar is on the class path. The rules described in "Case Sensitivity of Option Arguments" on page 4-6 apply. Table 4–5 lists the available options.

**Note:** SignatureTest command settings for the CLASSPATH environment variable and the -classpath option are the same as those listed for the Setup command in Table 4–1.
4.4.1 Command Description

The SignatureTest command compares the reference API represented in a signature file to the API under test and produces a report. Depending on the command-line options specified, it uses either the reflection or static mode. If the -classpath option is specified, the SignatureTest command checks if any extra classes are contained in the APIs it specifies.

See "Signature File Formats" on page 4-7 for details about supported versions of signature file formats.

**Note:** Arguments to options marked with an asterisk (*) in Table 4–5 can be specified multiple times. For example, if you want to specify filtering for the java.awt, java.lang, and java.math packages, you can specify the three arguments separated by a file separator character as shown in the following example: --Package java.awt:java.lang:java.math On Unix operating systems the separator character is “:” and on Microsoft Windows it is “;”.

<table>
<thead>
<tr>
<th>Table 4–5 SignatureTest Command Options</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>-help</td>
<td>Optional. Displays usage information for available command options and exits.</td>
</tr>
<tr>
<td>-debug</td>
<td>Optional. Enables printing of the stack trace for debugging purposes if SignatureTest fails.</td>
</tr>
<tr>
<td>-static</td>
<td>Optional. Specifies to run in static mode without using reflection and reports on only the class files specified in the -classpath option.</td>
</tr>
<tr>
<td>-mode [bin</td>
<td>src]</td>
</tr>
<tr>
<td>-CheckValue</td>
<td>Specifies to check the values of primitive and string constants. This option generates an error if a signature file does not contain the data necessary for constant checking.</td>
</tr>
<tr>
<td>-NoCheckValue</td>
<td>Specifies not to check the values of primitive and string constants.</td>
</tr>
<tr>
<td>-ClassCacheSize size_of_cache</td>
<td>Optional. Used in static mode only. Default is 1024. Specifies the size of the class cache as a number of classes to be held in memory to reduce execution time. Increasing this value dedicates more memory to this function.</td>
</tr>
<tr>
<td>-classpath path</td>
<td>Optional. Specifies the path to one or more APIs to be tested. Defaults to the classes contained in the signature file under test. Can contain multiple directories or ZIP or JAR files. The -package option filters the set of classes specified in -classpath (see &quot;CLASSPATH and -classpath Settings&quot; on page 4-3). Uses the path separator appropriate for the platform (identified by java.io.File.pathSeparator).</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-TestURL URL</td>
<td>Optional. Specifies the directory location of the signature file as a file protocol URL: \file://path \ Must end in a trailing forward slash on a UNIX system or a backslash on a Microsoft Windows or DOS system.</td>
</tr>
<tr>
<td>-FileName file_name</td>
<td>Required if -Files is not specified. Specifies the name of a signature file to be used.</td>
</tr>
<tr>
<td>-Files file_names</td>
<td>Required if -FileName is not specified. Use this option for testing a combination of APIs represented by corresponding signature files. Specifies the names of the signature files to be used delimited by a file separator. The file separator on UNIX systems is a colon (:) character, and on Microsoft Windows systems it is a semicolon (;). See &quot;Signature File Merge Rules&quot; on page 4-4 for details about the rules used for merging.</td>
</tr>
<tr>
<td>-NoMerge</td>
<td>Optional. Forces using the first encountered class description if more than one class with the same name is found in the input set of signature files specified by -Files option. In this case it uses the class from the left-most signature file that is specified with -Files. This option prevents the test from using the default merging behavior according to the JSR 68 rules. See &quot;Signature File Merge Rules&quot; on page 4-4 for details about the rules used for merging.</td>
</tr>
<tr>
<td>-Package package_or_class_name*</td>
<td>Optional. Specifies a class or package to be reported on, including its subpackages if a package is specified. The default is all classes and packages in the signature file. This option acts as a filter on the set of classes or packages optionally specified in -classpath. Repeat the option to specify multiple entries.</td>
</tr>
<tr>
<td>-PackageWithoutSubpackages package*</td>
<td>Optional. Similar to the -package option, specifies a package to be reported on without its subpackages. Repeat the option to specify multiple entries.</td>
</tr>
<tr>
<td>-Exclude package_or_class_name*</td>
<td>Optional. A package or class to be excluded from the report, including its subpackages. Repeat the option for multiple entries. Excludes duplicate entries specified by the -package or the -PackageWithoutSubpackages option.</td>
</tr>
<tr>
<td>-out file_name</td>
<td>Optional. Prints report messages to a specified file instead of standard output.</td>
</tr>
<tr>
<td>-FormatPlain</td>
<td>Optional. Specifies not to sort report messages, but output them immediately. Allows a decrease in memory consumption compared to the default sorted format of message reporting.</td>
</tr>
<tr>
<td>-FormatHuman -H</td>
<td>Optional. Specifies the human-readable report output format. Simple API changes are presented not as pair of errors (“missing element” and “added element”), but as a single API change.</td>
</tr>
</tbody>
</table>

Using the Signature Test Tool 4-11
### 4.4.2 Using Update Files

An update file is an XML file that contains API changes that can be applied to a signature file before tests are performed. An update can have an ID and corresponding comments. Example 4–1 shows an example of an update file that removes and adds a class before test execution.

#### Example 4–1 Update File Example

```xml
<updateset>
  <update type="removeclass" id="Change #12" comments="Fixing bug #1234"
    classname="com.acme.Elefant" />
  <update type="addclass" id="Change #13" comments="Approved by the committee"
    classname="com.acme.Tiger" />
  CLSS public com.acme.Tier">
    cons public <init>()
    cons public <init>(java.lang.String)
    supr java.lang.Object
  </update>
</updateset>
```

The XML file is enclosed within the `<updateset>` element and each change is specified in an `<update>` element. Each `<change>` element consists of a type and a set of attributes. The types and their attributes are described in Table 4–6.

---

**Table 4–5 (Cont.) SigTest Command Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Backward</td>
<td>Optional. Specifies migration compatibility check mode.</td>
</tr>
<tr>
<td>-B</td>
<td></td>
</tr>
<tr>
<td>-apiVersion version_string</td>
<td>Optional. Specifies the API version number of the implementation under test to be recorded in the report.</td>
</tr>
</tbody>
</table>
| -verbose [nowarn | noerr]               | Optional. Enables error diagnostics for inherited class members. By default, such error messages are suppressed.  
                       | nowarn: Suppress warnings.  
<pre><code>                   | noerr: Suppress errors. |
</code></pre>
<table>
<thead>
<tr>
<th>Update Types</th>
<th>Description</th>
</tr>
</thead>
</table>
| removeclass  | Removes a class. Required attributes:  
  - classname - the name of the class to remove  
Optional attributes:  
  - id - change ID  
  - comments - a comment that describes the change  
Example:  
```xml  
<update type="removeclass" classname="com.acme.Zoo" id="Issue #1717" comments="Bug fixing" />  
``` |
| removepackage | Removes all classes belonging to a specified package. Required attributes:  
  - packagename - the name of the package to remove  
Optional attributes:  
  - id - change ID  
  - comments - a comment that describes the change  
Example:  
```xml  
<update type="removepackage" packagename="com.acme.animals" id="CR 4657" comments="Removing incorrectly introduced package, approved" />  
``` |
| addclass     | Adds a class. Required attributes:  
  - classname - the name of the class to remove  
  - Definition of the new class in the element body  
Optional attributes:  
  - id - change ID  
  - comments - a comment that describes the change  
Example:  
```xml  
<update type="addclass" classname="com.acme.zoo.Rabbit">  
CLSS public com.acme.zoo.Rabbit  
cons public <init>()  
cons public <init>(java.lang.String)  
supr com.acme.zoo.Animal  
</update>  
```
You can cause SignatureTest command report messages to be sorted (default), or unsorted by specifying the -FormatPlain flag at the command line. See Step 8 of the procedure “Using the SetupAndTest Command” on page A-4 to see a sorted report that was generated into a plain text file with the -out option. Report messages contain the following types of information:

- The versions of both the reference API and the API under test
- The total number of errors found
- The differences such as added or missing classes, superclasses, fields, or methods
- The fully qualified name of the enclosing class related to any missing or added description
- Two errors for any modified item, one for a missing item and another for an added item
- A description of any incompatibility in the API implementation under test

4.4.3.1 Sorted Report
Report messages are sorted by default. Unlike the unsorted format, duplicate messages are removed. To accurately compare the error totals between formats, the sorted report prints two error counts: the total number of errors and the total number of duplicates.

The sorted report groups error messages by category of difference with classes within each category ordered alphabetically by name, and empty categories are ignored. This is the category ordering sequence:

- Missing Classes
- Missing Class Descriptions (Modified classes and nested classes)
- Missing Superclasses or Superinterfaces
- Missing Fields
- Missing Constructors
- Missing Methods
- Added Classes
- Added Class Descriptions (Modified classes and nested classes)
- Added Superclasses or Superinterfaces
- Added Fields
- Added Constructors
- Added Methods
- Linkage Errors

Linkage errors occur during testing if the API implementation under test is corrupted, for example, if a superclass or superinterface of a class under test cannot be loaded.

See the sorted report in Example A–4.

4.4.3.2 Unsorted Report
The -FormatPlain option specifies an unsorted report. The unsorted format reports messages immediately during execution and duplicate messages are included.

Example 4–2 shows an unsorted report that corresponds to the sorted report in Example A–4.

Example 4–2  Unsorted Report Example
Definition required but not found in example.test
   method public int get(int)
Definition found but not permitted in example.test
4.4.3.3 Human-Readable Report

The human-readable report format is similar to the sorted report format. Unlike the sorted report format where some changes in the API are presented as pairs of messages (“added element” and “missed element”), the human-readable report format presents these changes as a single message that describes the difference between the APIs. This report recognizes the following cases:

- Modifiers were changed
- Return type was changed
- Parameter types were changed
- Normalized throw list was changed
- Constant value was changed
- Annotation was changed

Unlike the sorted report format, all messages are grouped by class first, rather than message type. The code example below shows the output for a class:

```java
Class tests.human.many.E1
  Changed
  method public abstract void tests.human.many.E1.foo() throws java.io.IOException
  - Throws: [java.io.IOException]
  method public abstract void tests.human.many.E1.foo() throws java.io.IOException
  - [public, abstract]
  + [protected]
  field public final static int tests.human.many.E1.i = 0
  - type: int
  + type: char
  field public final static int tests.human.many.E1.i = 0
  - value: 0
```

4.5 SetupAndTest Command

The SetupAndTest command has the following synopsis:

```
java -jar sigtestdev.jar SetupAndTest [options]
```

or

```
java com.sun.tdk.signaturetest.SetupAndTest [options]
```

The second alternative assumes that sigtestdev.jar is on the class path. The rules described in "Case Sensitivity of Option Arguments" on page 4-6 apply. Table 4–7 lists the available options.

4.5.1 Command Description

SetupAndTest is a wrapper command that runs only in static mode to execute the Setup and SignatureTest commands in one operation. It performs these functions:
1. Calls the Setup command to create a signature file from the reference API specified as input. It creates a temporary signature file if no name is specified for it with the -FileName option.

2. Calls the SignatureTest command to make the comparison and produce a report.

**Note:** Arguments to options marked with an asterisk in Table 4–7 can be specified multiple times. For example, if you want to specify filtering for the java.awt, java.lang, and java.math packages, you can specify the three arguments separated by a file separator character as shown in the following example: –Package java.awt:java.lang:java.math On Unix operating systems the separator character is “:” and on Microsoft Windows it is “;”.

### Table 4–7 SetupAndTest Command Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-help</td>
<td>Optional. Displays usage information for available command options and exits.</td>
</tr>
<tr>
<td>-reference path</td>
<td>Required. Specifies the path to the classes to be used for creating the signature file.</td>
</tr>
<tr>
<td>-test path</td>
<td>Required. Path to the packages being tested.</td>
</tr>
<tr>
<td>-FileName file</td>
<td>Optional. Specifies the signature file name. The command execution uses a temporary file if not specified.</td>
</tr>
<tr>
<td>-mode [bin</td>
<td>src]</td>
</tr>
<tr>
<td>-package package_or_class_name*</td>
<td>Optional. Specifies a class or package to be reported on, including its subpackages if a package is specified. This option acts as a filter on the set of classes or packages that are tested and reported on. The default is all classes and packages. Repeat the option to specify multiple entries.</td>
</tr>
<tr>
<td>-PackageWithoutSubpackages name*</td>
<td>Optional. Specifies package to be tested excluding subpackages.</td>
</tr>
<tr>
<td>-exclude name*</td>
<td>Optional. A package or class to be excluded from testing, including its subpackages. Repeat the option for multiple entries. Excludes duplicate entries specified by the -package or the -PackageWithoutSubpackages option.</td>
</tr>
<tr>
<td>-ClassCacheSize size_of_cache</td>
<td>Optional. Specifies the size of the class cache as a number of classes to be held in memory to reduce execution time. Increasing this value dedicates more memory to this function. Defaults to 100.</td>
</tr>
<tr>
<td>-CheckValue</td>
<td>Optional. Specifies to check the values of primitive and string constants. This option generates an error if a signature file does not contain the data necessary for constant checking.</td>
</tr>
</tbody>
</table>
The Merge command has the following synopsis:

```
java -jar sigtest.jar Merge [options]
```

or

```
java com.sun.tdk.signaturetest.Merge [options]
```

The second alternative assumes that `sigtest.jar` is on the class path. The rules described in “Case Sensitivity of Option Arguments” on page 4-6 apply. Table 4-8 lists the available options.

### 4.6.1 Command Description

The Merge command combines (merges) a number of input signature files into one resulting signature file. See "Signature File Merge Rules" on page 4-4 for details about the rules used for merging, and "Signature File Formats" on page 4-7 for details about supported versions of signature file formats. Table 4-8 lists the Merge command options.
### Table 4–8  Merge Command Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-help</td>
<td>Optional. Displays usage information for available command options and exits.</td>
</tr>
<tr>
<td>-Files</td>
<td>Required. Specifies the names of the signature files to be merged delimited by a file separator. The file separator on UNIX systems is a colon (:) character, and on Microsoft Windows systems it is a semicolon (;). See &quot;Signature File Merge Rules&quot; on page 4-4 for details about the rules used for merging.</td>
</tr>
<tr>
<td>-Write</td>
<td>Required. Specifies the resulting output signature file.</td>
</tr>
<tr>
<td>-Binary</td>
<td>Optional. Specifies to use the binary merge mode. See &quot;Source and Binary Compatibility Modes&quot; on page 3-4.</td>
</tr>
</tbody>
</table>
Describes how you can use the API Coverage tool to estimate the test coverage a test suite provides for an implementation of a specified API. It does this by determining how many public class members the test suite references within the API specification. The tool uses a signature file representation of the API specification as the source of specification analysis. It does not process a formal specification in any form.

This part includes the following chapters:

- Chapter 5, "Introduction to API Coverage Tool"
- Chapter 6, "Using the API Coverage Tool"
The API Coverage tool is used to estimate the test coverage that a test suite under development affords to implementations of its related API specification. It does this by determining how many public class members the test suite references within the API specification that it is designed to test against. The tool uses a signature file representation of the API specification as the source of specification analysis. It does not process a formal specification in any form.

The API Coverage tool code is contained in both the sigtestdev.jar and sigtest.jar files. Additional installation is not required. See Chapter 6 for details about running API Coverage tool.

5.1 Static API Coverage Analysis

The tool operates on the principle that a reference to an API member from within a test class indicates a test of that member by the test suite. The ratio of referenced class members to the total number of class members calculated for an API yields a percentage of test coverage for the API.

This method is called static API coverage analysis because it does not actually run any tests from the test suite. Since it does not dynamically determine which API members are actually accessed by the tests, the coverage calculation expresses only an estimated percentage of test coverage.

5.1.1 Major Source of Error

Static analysis cannot correctly predict the outcome of virtual calls to overridden methods that are resolved at runtime through dynamic method dispatch. The frequency of this type of overridden method can vary between differing implementations of the same API specification. This makes it difficult to formulate an exact percentage of test suite coverage when using static analysis (in spite of the fact that the implementations may all be binary compatible and correctly implement the specification).

Tests that make dynamic calls to API members are not recognized by API Coverage tool. This may cause some test calls to not be accurately measured and may result in test coverage being underreported.

5.1.2 Advantages of Static Coverage Analysis

Static API coverage testing provides the following advantages over dynamic coverage methods.
Static testing is more lightweight. Testing is easier to setup and quicker to run than dynamic methods.

Static testing is easier to automate and provides more consistent results because it is not affected by external conditions such as machine load, or network traffic.

Static testing is more practical for gathering results for very large APIs. A test suite and its associated API might include many thousands of tests and associated API class members making it very cumbersome to instrument dynamic tests.

Static testing allows you to quickly estimate the quality of test coverage for all APIs, especially APIs that are difficult to test dynamically.

5.2 How It Works

The static API coverage algorithms examine precompiled test suite test classes to determine the members that they reference. This includes inner classes and fields as well as constructors, although constructors cannot be inherited.

The algorithms are based on the fact that the constant pool of any class file holds all of its external class references. This constant pool consists of the following related records:

- CONSTANT_Fieldref
- CONSTANT_Methodref
- CONSTANT_InterfaceMethodref

Each of these records contains the fully qualified name of an external class and the name of the class member referenced. For a method, this includes the signature of the method and its return type.

5.2.1 Level of Accuracy During Analysis

Table 5–1 lists example scenarios encountered during static coverage analysis and their related potential for error. The table references the following objects:

- $Q.m$ is a method referenced by the test suite.
  - Where $Q$ is the fully qualified class name, and $m$ is the descriptor of the called method (including the name, list of arguments, and return type).
- $SubQ$ is a subclass of, and $SupQ$ is a superclass of class $Q$.
- $x$ is an object on the stack referenced by either the `invokevirtual` or `invokeinterface` instruction.

The main potential for an inaccurate coverage measurement exists when a great many members are overridden in subclasses of an implementation (as described in condition #2 of Table 5–1).

<table>
<thead>
<tr>
<th>Condition</th>
<th>Scenario</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1: Object $x$ is of type $Q$</td>
<td>Class $Q$ declares method $m$ and $Q.m$ is the method called</td>
<td>$Q.m$ is correctly marked as covered</td>
</tr>
<tr>
<td></td>
<td>Method $m$ is inherited from superclass $SupQ$ and is not declared in $Q$; method $SupQ.m$ is called</td>
<td>Correctly marks either $Q.m$ or $SupQ.m$ as covered, depending on the calculation mode in use</td>
</tr>
</tbody>
</table>

(Described later in "Coverage Analysis Modes" on page 5-3)
5.2.2 Coverage Analysis Modes

The API Coverage tool uses these two modes of analysis:

- **Real World Mode**: Returns a fairly accurate estimate based on input from one specific API implementation, such as a reference implementation. You can then compare the real world results to those of the worst case mode.

- **Worst Case Mode**: Returns an estimate based on a hypothetical API in which every class overrides or hides all members from its superclass. This scenario is highly unlikely to occur in actual practice. You use this mode by extrapolating its results into those of the real world case to estimate the possible range in test coverage that a test suite will provide in the field over a number of differing implementations.

Chapter 6 describes how to set up and use the API Coverage tool.

5.2.3 Filtering Coverage By Marking Up Signature Files

Typically, API Coverage tool measures coverage for an entire API or package(s). However, in some use cases it is convenient to track the coverage of a particular feature that spreads throughout multiple packages and classes.

For example, you might want to track a new feature that adds a few methods or fields to preexisting classes, or you might want to track a feature that adds an extra interface implemented by a preexisting class. You will want to know which of those newly added methods are tested.

The way to get this information is to mark up a signature file with directives that indicate which API members belong to the feature that you want reported in the API Coverage tool report. A fragment of a signature file with filtering directives (in bold) is shown in Example 5–1.

**Example 5–1  Signature File Mark-Up**

```
#APICover file v4.1
CLSS public final java.awt.SplashScreen
meth public boolean isVisible()
  #coverage on
meth public java.awt.Dimension getSize()
meth public java.awt.Graphics2D createGraphics()
  #coverage off
```

<table>
<thead>
<tr>
<th>Condition</th>
<th>Scenario</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>#2: Object x is of type SubQ, a subclass of Q</td>
<td>No subclass or superclass of Q overrides method m; Q.m is called</td>
<td>Correctly marks Q.m as covered</td>
</tr>
<tr>
<td></td>
<td>A subclass of Q does override method m and the overriding method is called; if there are multiple inherited subclasses, exactly which method is called cannot be correctly identified before runtime</td>
<td>Q.m is incorrectly marked as covered; this scenario is the main source of analysis errors</td>
</tr>
<tr>
<td>#3: A method is called by means of reflection</td>
<td>Uses: Method.invoke(Object, Object[])</td>
<td>No method is marked as covered, assuming that java.lang.reflect is not in the API under test; this case cannot be correctly resolved</td>
</tr>
</tbody>
</table>

Table 5–1 (Cont.) Example Scenarios and Potential Errors
meth public java.awt.Rectangle getBounds()
meth public java.net.URL getImageURL()
meth public static java.awt.SplashScreen getSplashScreen()
meth public void close()
meth public void setImageURL(java.net.URL) throws java.io.IOException
meth public void update()
supr java.lang.Object
hfds image,imageURL,log,splashPtr,theInstance,wasClosed

5.2.3.1 Filtering Markup Format
The following steps describe how to mark up a signature file to designate classes and members you want to track.
1. Copy the signature file.
2. In a text editor, change the first line of the copy of the signature file from #Signature file v4.1 to #APICover file v4.1.
3. Use the #coverage on and #coverage off directives to mark the classes and members.
   Classes or members between the #coverage on and #coverage off directives are tracked during coverage calculation. All the other classes and members are excluded.
Using the API Coverage Tool

API Coverage tool requires these environmental settings and components:

- A properly configured certified Java Platform, Standard Edition (“Java SE platform”) runtime version 1.4 or later
- One of the following JAR files in your class path:
  - sigtestdev.jar
  - sigtest.jar

The API Coverage tool Main class is contained in both JAR files.

The API Coverage tool is a command-line utility that executes in a Java application environment. It generates static API test coverage reports in either plain text or XML format based on the following two input items:

- The test suite being tested for its coverage, represented by a set of test class files
- A signature file that forms a text representation of the public members of an API that must be tested by the test suite

You can use the Setup command of the Signature Test tool to create the appropriate signature file for use by the API Coverage tool. The Setup command is described in Chapter 4.

6.1 Running API Coverage Tool

API Coverage tool command synopsis:

% java -jar apicover.jar [options]
or
% java com.sun.tdk.apicover.Main [options]

The second alternative assumes that apicover.jar is on the class path. The rules described in "Case Sensitivity of Option Arguments" on page 4-6 apply. Table 6–1 lists the available options.

---

Note: Arguments to options marked with an asterisk (*) in Table 6–1 can be specified multiple times. For example, if you want to specify filtering for the java.awt, java.lang, and java.math packages, you can specify the three arguments separated by a file separator character as shown in the following example: -tsInclude java.awt:java.lang:java.math On Unix operating systems the separator character is “:” and on Microsoft Windows it is “;”.

---
### Table 6–1  API Coverage Tool Command Options

<table>
<thead>
<tr>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-help</td>
<td>Optional. Displays usage information for available command options and exits.</td>
</tr>
<tr>
<td>-ts path</td>
<td>Required. Cannot be repeated. Path to the test suite classes to be analyzed. Also accepts a JAR file.</td>
</tr>
<tr>
<td>-tsInclude package_name*</td>
<td>Optional. Can be repeated. Recursively include classes from the specified test suite package.</td>
</tr>
<tr>
<td>-tsIncludeW package_name*</td>
<td>Optional. Can be repeated. Include classes from the specified test suite package without subpackages.</td>
</tr>
<tr>
<td>-tsExclude package_name*</td>
<td>Optional. Can be repeated. Recursively exclude classes from the specified test suite package.</td>
</tr>
<tr>
<td>-excludeList filename</td>
<td>Optional. Can be repeated. Specifies the file name of an exclude list which contains API elements to be excluded from the coverage calculation. &quot;Exclude List&quot; on page 6-3 describes the exclude list format.</td>
</tr>
<tr>
<td>-excludeInterfaces</td>
<td>Optional. Exclude all interface classes. This option is superfluous if used with -excludeAbstractClasses.</td>
</tr>
<tr>
<td>-excludeAbstractClasses</td>
<td>Optional. Exclude all abstract classes including interfaces.</td>
</tr>
<tr>
<td>-excludeAbstractMethods</td>
<td>Optional. Exclude all abstract methods from all classes and interfaces.</td>
</tr>
<tr>
<td>-excludeFields</td>
<td>Optional. Exclude all fields from all classes and interfaces. cannot be used with the -includeConstantField option.</td>
</tr>
<tr>
<td>-includeConstantFields</td>
<td>Optional. Specifies to include all final fields of any type in the coverage calculation. cannot be used with the -excludeFields option. The default, without this option, is to remove from the coverage calculation all final fields of primitive types and type java.lang.String. Constant fields such as final fields of primitive and String types are initialized at compile-time with a constant expression. The Java technology-based compiler usually optimizes them and replaces them with their values. In such cases, test suite classes does not contain a reference to the field, even if the field were used/referenced in the test suite. For this reason, the tool removes all constant fields from the calculation by default to make coverage calculations more consistent. The -includeConstantFields option provides a means for changing this behavior.</td>
</tr>
<tr>
<td>-api path/filename</td>
<td>Required. Cannot be repeated. Specifies the location of the signature file representing the API under examination. The path defaults to the working directory. Accepts v1 and v2 format signature files generated by the Signature Test Tool Setup command, and rejects v0. See the -FileFormat option description in Chapter 4.</td>
</tr>
<tr>
<td>-apiInclude package_name*</td>
<td>Optional. Can be repeated. Recursively include classes from the specified API package.</td>
</tr>
<tr>
<td>-apiIncludeW package_name*</td>
<td>Optional. Can be repeated. Include classes from the specified API package without subpackages.</td>
</tr>
<tr>
<td>-apiExclude package_name*</td>
<td>Optional. Can be repeated. Recursively excludes classes from the specified API package.</td>
</tr>
</tbody>
</table>
Running API Coverage Tool

Using the API Coverage Tool

Table 6–2 describes the reporting levels resulting from the five settings available with the -detail option.

<table>
<thead>
<tr>
<th>Detail Setting</th>
<th>Package Summary</th>
<th>Class Summary</th>
<th>Members Not Covered</th>
<th>Covered Members</th>
<th>Reference Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>5</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

6.1.1 Special Report File

The special report file specified with the -out option contains the same coverage data that the standard report file contains. The main difference is that this format is easier for other applications to parse.

The special report file is essentially a table with these two columns delimited by a tab character:

- The left column provides the coverage counter value which can be zero.
- The right column provides the fully qualified name of the corresponding API member: a class, constructor, method, or field.

The -mode option affects the special report file, while the -detail setting has no effect on it.

6.1.2 Exclude List

In some cases a test suite either cannot or should not test certain elements of an API, whether they be a class, a method, or field. In this case you can use the -excludeList filename option (described in Table 6–1) to make the coverage calculation more accurate by excluding the elements listed in the specified exclude list file(s).
The exclude list is a text file that specifies each entry for exclusion on a separate line, as follows:

- **Package name** specified to recursively exclude all classes of the package
- **Class name** specified as a fully qualified class name
- **Field name** specified using a fully qualified class name: `classname.fieldname`
- **Method name** specified using a fully qualified class name: `classname.methodname (parameters)`

Exclude list processing follows these rules:

- Regards a line starting with the `#` character as a comment.
- Ignores empty lines.
- Silently ignores an entry if it is not found in the API.

---

**Note:** Each entry specified for exclusion should match its appearance in a report file. For example, you must specify an inner class with the `$` character prepended to its name, like this: `Outer$Inner`

---

### 6.2 Merging API Coverage XML Reports

The API Coverage Tool Merge command merges multiple API Coverage reports in XML format into a single report. Synopsis:

```
% java com.sun.tdk.apicover.CMerge [options]
```

The rules described in "Case Sensitivity of Option Arguments" on page 4-6 apply to the Merge command. Table 6–3 lists the Merge command options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Files</td>
<td>Required. Specifies the names of the API coverage XML report files to be merged. The file name are separated by a colon (:) on Unix systems, and by a semicolon (;) on Microsoft Windows systems.</td>
</tr>
<tr>
<td>Write</td>
<td>Required. Specifies the merged output report file.</td>
</tr>
<tr>
<td>Strict</td>
<td>Optional. Requires identical set of members for the same classes.</td>
</tr>
<tr>
<td>Version</td>
<td>Optional. Displays version information, then exits.</td>
</tr>
<tr>
<td>Help</td>
<td>Optional. Displays usage information for available command options, then exits.</td>
</tr>
</tbody>
</table>
This appendix describes how to use the API Check tool to track API changes and check for source and binary compatibility.

This part includes the following chapters:

- Chapter 7, "Introduction API Check Tool"
- Chapter 8, "Using the API Check Tool"
This chapter introduces the API Check Tool. Chapter 8 describes how to use the tool.

7.1 The API Check Tool

The API Check tool tracks API changes and checks for source and binary compatibility. Much like Signature Test tool, it compares the reference implementation of an API recorded as golden signature file with a tested implementation. Unlike Signature Test tool, which requires that all classes (and the classes they depend on) be specified to the tool, API Check tool does not require dependencies to be specified. For that reason, API Check tool is faster than Signature Test tool, but less rigorous.

As an example of how this can be useful, consider the case of a developer who uses the JavaMail API. The developer might wish to check whether a new implementation of the JDK software contains all members that the JavaMail API calls — essentially a quick signature check that looks for missing signatures. API Check tool provides a fast and simple way to generate a report to show this.
API Check tool behaves much the same as Signature Test tool and uses the same options and signature files. See Chapter 4 for a description of the options. The input signature file is generated by the Signature Test tool Setup command using the -NonClosedFile option. Neither the CLASSPATH variable used for reflection mode, nor classes listed with -classpath option for static mode can contain base classes. See "CLASSPATH and -classpath Settings" on page 4-3 for more information.

8.1 API Check Tool Verification Scope

API Check tool verifies classes in signature files that are also covered by packages defined using the -package option. If the specified signature file is generated using the -NonClosedFile option, all superclasses and superinterfaces outside this scope are not checked.

As an example, consider a signature file that contains java.util classes. Use the following command to create the signature file:

```
java -jar sigtestdev.jar setup -classpath $JAVA_HOME/jre/lib/rt.jar
   -filename java-util.sig -nonclosedfile -package java.util
```

Use the following command to verify the java.util.ArrayList class using the signature file created in the previous command:

```
java -jar apicheck.jar -static -classpath $JAVA_HOME/jre/lib/rt.jar -filename java-util.sig -package java.util.ArrayList
```

Because java.util.ArrayList is specified as the scope for checking, it is the only class verified. ArrayList has a branching class and interface hierarchy that belongs to the java.lang, java.util and java.io packages as shown in Figure 8–1. Because the input signature file only contains classes from the java.util package, only the ArrayList class superclasses and superinterfaces from java.util are analyzed and their members are checked as ArrayList class inherited members.
Consider a similar signature file that contains java.lang classes. In a similar check for the RuntimePermission class, the RuntimePermission hierarchy starts in the java.lang.Object class, departs the java.lang package to java.security and finally returns to the java.lang package:

java.lang.Object
    extended by java.security.Permission
    extended by java.security.BasicPermission
    extended by java.lang.RuntimePermission

API Check tool cannot verify any of the RuntimePermission class hierarchy because its direct superclass is outside the scope. In this case, even though java.lang.Object is in scope no inherited members are taken into account, and only members declared in RuntimePermission itself are analyzed.

8.2 Running API Check Tool

API Check tool command synopsis:

% java -jar apicheck.jar [options]
or
% java com.sun.tdk.apicheck.Main [options]

The rules described in "Case Sensitivity of Option Arguments" on page 4-6 apply.
The command options are the same as those for Signature Test Tool and are listed in Table 4–5.
The settings for the CLASSPATH environment variable and the -classpath option are the same as those for Signature Test Tool and are listed in Table 4–1.
This part contains appendices that include step-by-step examples that show how to use the tools, API migration rules, and information about using the tools with Ant and Maven.

This part includes the following appendices:

- Appendix A, "Signature Test Tool Quick Start Examples"
- Appendix B, "API Coverage Tool Quick Start Examples"
- Appendix C, "API Migration Compatibility Rules (Signature Test)"
- Appendix D, "Merge Command Operative Principles"
- Appendix E, "Using the SigTest Tools With Ant"
- Appendix F, "Using the SigTest Tools With Apache Maven"
- Section G, "Custom SigTest Tool Example"
This appendix contains the following two step-by-step examples that show you how to use the Signature Test tool:

- Example 1: Compare Two Different Implementations of the Same API
- Example 2: Merge Two Signature Files

### A.1 Example Programs

Sample sources used in the examples, along with simple UNIX operating system shell and Windows batch scripts are provided in the `examples/userguide` directory. You can use these to help run the examples. The README file in that directory explains how to use them.

#### Note:
The examples use UNIX operating system syntax.

#### Caution:
Code examples are an optional part of the SigTest distribution and are not required to use the SigTest tools. You can remove the example directory (`$SIGTEST_HOME/examples`) if you are concerned about security issues the examples might create.

Example A–1 and Example A–2 show the two implementations of the test API used throughout these examples.

**Example A–1  Version 1.0 of test.java**

```java
package example;
public class test {
    public <T> T get (T t) {
        return t;
    }
}
```

**Example A–2  Version 2.0 of test.java**

```java
package example;

public class test {
    public String get (int i) {
```
public void put () {
    }
}

A.2 Example 1: Compare Two Different Implementations of the Same API

You can compare two implementations of an API in two ways:

- Run the Setup command to create a signature file for the first implementation and then run the SignatureTest command to compare them. This is shown in the first part of this example: A.2.1.
- Run the SetupAndTest command. The command combines the Setup and SignatureTest commands. This is shown in the second part of this example: A.2.2.

A.2.1 Using the Setup and SignatureTest Commands

This example shows how to compare the following implementations of the test.java API using the Setup and SignatureTest commands.

1. Make SigTest-directory/examples/usersguide the current directory.
2. Compile the two implementation files.
   SigTest tool works with class files. Use the following commands to compile the source files into class file:
   
   % javac -d V1.0 V1.0/test.java
   % javac -d V2.0 V2.0/test.java

3. Set the CLASSPATH environment variable to include the path to sigtestdev.jar.
   
   % setenv CLASSPATH=../../lib/sigtestdev.jar

4. Set the JAVA_HOME environment variable to include the base directory of the Java platform runtime installation on your system.
   
   % setenv JAVAHOME=java_runtime_path

5. Use the Setup command to create a signature file for the V1.0 implementation.
   Note that the "\" characters are used to break long lines. Type this as a single line at the command line.
   
   % java com.sun.tdk.signaturetest.Setup \
   -classpath V1.0:$JAVA_HOME/jre/lib/rt.jar \
   -static \
   -apiVersion V1.0 \
   -package example \
   -fileName test.sig

   The Setup command emits console output similar to the following:
   
   Constant checking: on
   Found in total: 16260 classes
   Selected by -Package: 1 classes
   Written to sigfile: 2 classes
   STATUS:Passed.
6. Confirm that the contents of the signature file are correct by comparing it to the following file:

```
#Signature file v4.0

CLSS public example.test
  cons public <init>()
  meth public <%0 extends java.lang.Object> {%%0} get({%%0})
  supr java.lang.Object

CLSS public java.lang.Object
  cons public <init>()
  meth protected java.lang.Object clone() throws java.lang.CloneNotSupportedException
  meth protected void finalize() throws java.lang.Throwable
  meth public boolean equals(java.lang.Object)
  meth public final java.lang.Class<?> getClass()
  meth public final void notify()
  meth public final void notifyAll()
  meth public final void wait() throws java.lang.InterruptedIOException
  meth public final void wait(long) throws java.lang.InterruptedIOException
  meth public int hashCode()
  meth public java.lang.String toString()
```

7. Use the `SignatureTest` command to compare the V2.0 API signature with the V1.0 signature file you created in Step 5.

```
% java com.sun.tdk.signaturetest.Setup \
   -classpath V2.0:$JAVA_HOME/jre/lib/rt.jar \
   -static \
   -package example \
   -fileName test.sig \
   -out report.txt
```

The `-out` option directs the command to write its results to `report.txt`.

A message similar to the following is written to the console when the command terminates:

```
See log recorded to file report.txt for more details.
STATUS:Failed.3 errors
```

8. Confirm that the contents of the `report.txt` file are correct by comparing it to the following file:

```
SignatureTest report
Tested version:
Check mode: src [throws normalized]
Constant checking: off

Missing Methods
-----------------
example.test: method public <%0 extends java.lang.Object> {%%0} example.test.get({%%0})
```

**Note:** There might be some minor differences because of system-specific information.
Added Methods
-------------
example.test: method public java.lang.String example.test.get(int)
example.test: method public void example.test.put()

STATUS:Failed.3 errors

A.2.2 Using the SetupAndTest Command

The SetupAndTest command combines the functionality of the Setup and SignatureTest commands into a single command.

1. Follow steps 1 - 4 in the previous example.
2. Use the following SetupAndTest command to create a signature file.

   % java $CLASSPATH com.sun.tdk.signaturetest.SetupAndTest \
   -reference V1.0:$JAVA_HOME/jre/lib/rt.jar \
   -test V2.0:$JAVA_HOME/jre/lib/rt.jar \
   -package example

   The SetupAndTest command reports results similar to following example to standard output.

   Invoke Setup ...
   Class path: "V1.0;C:\java\jdk1.5.0_06\jre\lib\rt.jar"
   Constant checking: on
   Found in total: 12749 classes
   Selected by -Package: 1 classes
   Written to sigfile: 2 classes
   Invoke SignatureTest ...
   SignatureTest report
   Tested version: 2.0
   Check mode: src [throws normalized]
   Constant checking: on

Missing Methods
---------------
example.test: method public <%0 extends java.lang.Object> {%%0} example.te st.get({%%0})

Added Methods
-------------
example.test: method public java.lang.String example.test.get(int)
example.test: method public void example.test.put()

STATUS:Failed.3 errors

A.3 Example 2: Merge Two Signature Files

This example shows how to use the Merge command to accomplish the following:

1. Compile three .java files to produce .class files.
2. Run the Setup command on each .class file to produce a signature file.
3. Use the Merge command to combine the files.

A.3.1 Running Merge Example

These examples use the environment variables listed below:

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASSPATH</td>
<td>Must include the sigtestdev.jar file. For example: /sigtest-1.5.0_09/lib/sigtestdev.jar</td>
</tr>
<tr>
<td>RT_JAR</td>
<td>The location of the runtime rt.jar file. For example: /opt/jdk1.5.0_09/jre/lib/rt.jar</td>
</tr>
</tbody>
</table>

1. Set up the environment with the environment variable settings in the table above.
2. Create each of the files shown in the following examples. Create each in a different directory but with the same file name.

   Create directories named 1, 2, and 3. Add the appropriate file to each directory.

   ```java
   package x;
   public class A {
       public void abc() {}
       public void foo() {}
   }
   ```

   ```java
   package x;
   public class A {
       public void abc() {}
       public void bar() {}
   }
   ```

   ```java
   package x;
   public class A {
       public static void abc() {}
   }
   ```

3. Run these commands to compile each file into a separate x subdirectory:

   ```bash
   % javac -d 1 1/A.java
   % javac -d 2 2/A.java
   % javac -d 3 3/A.java
   ```

   This creates the following class files:

   a. ./1/x/A.class
   b. ./2/x/A.class
   c. ./3/x/A.class

4. Run the following three Setup commands on each A.class file to produce the three x#.sig files like the ones shown in following examples.

   a. Command #1:

   ```bash
   % java -cp 1:$CLASSPATH com.sun.tdk.signaturetest.Setup \ 
   -static -classpath 1:$RT_JAR -package x -FileName x1.sig
   ```
The command generates the ./x1.sig file like the one shown in the following example and produces a similar console message.

Class path: "1:/opt/jdk1.5.0_09/jre/lib/rt.jar"
Constant checking: on
Found in total: 13185 classes
Selected by -Package: 1 classes
Written to sigfile: 2 classes
STATUS:Passed.

b. Command #2:

% java -cp 2:$CLASSPATH com.sun.tdk.signaturetest.Setup \
-static -classpath 2:$RT_JAR -package x -FileName x2.sig

The command generates the ./x2.sig file shown in the following example and produces a console message similar to this indicating successful setup.

Class path: "2:/opt/jdk1.5.0_09/jre/lib/rt.jar"
Constant checking: on
Found in total: 13185 classes
Selected by -Package: 1 classes
Written to sigfile: 2 classes
STATUS:Passed.

c. Command #3:

% java -cp 3:$CLASSPATH com.sun.tdk.signaturetest.Setup \
-static -classpath 3:$RT_JAR -package x -FileName x3.sig

The command generates the ./x3.sig file shown in following example and produces a console message similar to this indicating successful setup.

Class path: "3:/opt/jdk1.5.0_09/jre/lib/rt.jar"
Constant checking: on
Found in total: 13185 classes
Selected by -Package: 1 classes
Written to sigfile: 2 classes
STATUS:Passed.

5. Run the following command to merge x1.sig and x2.sig and produce the x1+x2.sig file shown in the following example:

% java -cp $CLASSPATH com.sun.tdk.signaturetest.Merge -Files x1.sig:x2.sig -Write x1+x2.sig

The command generates a console message similar to the following message:

Warning: class java.lang.Throwable not found
STATUS:Passed.

6. Run this command to merge x2.sig and x3.sig attempting to produce the x2+x3.sig file:

% java -cp $CLASSPATH com.sun.tdk.signaturetest.Merge -Files x2.sig:x3.sig -Write x2+x3.sig

The command prints a message to the console similar to the following. it indicates a conflicting static modifier, and no signature file is created:

x.A.abc : <static> modifier conflict
STATUS:Error.Error
The conflict occurs because x2.sig contains this method (see Example A–4):

meth public void x.A.abc()
and x3.sig contains this method:
meth public static void x.A.abc()

### A.3.2 Example Result Files

This section contains the files that are generated from the previous Merge examples.

**Example A–3  Contents of ./x1.sig**

```plaintext
#Signature file v4.0
#Version

CLSS public java.lang.Object
cons public <init>()
meth protected java.lang.Object clone() throws java.lang.CloneNotSupportedException
meth protected void finalize() throws java.lang.Throwable
meth public boolean equals(java.lang.Object)
meth public final java.lang.Class<?> getClass()
meth public final void notify()
meth public final void notifyAll()
meth public final void wait() throws java.lang.InterruptedIOException
meth public final void wait(long) throws java.lang.InterruptedIOException
meth public final void wait(long,int) throws java.lang.InterruptedIOException
meth public int hashCode()
meth public java.lang.String toString()

CLSS public x.A
cons public <init>()
meth public void abc()
meth public void foo()
supr java.lang.Object
```

**Example A–4  Contents of ./x2.sig**

```plaintext
#Signature file v4.0
#Version

CLSS public java.lang.Object
cons public <init>()
meth protected java.lang.Object clone() throws java.lang.CloneNotSupportedException
meth protected void finalize() throws java.lang.Throwable
meth public boolean equals(java.lang.Object)
meth public final java.lang.Class<?> getClass()
meth public final void notify()
meth public final void notifyAll()
meth public final void wait() throws java.lang.InterruptedIOException
meth public final void wait(long) throws java.lang.InterruptedIOException
meth public final void wait(long,int) throws java.lang.InterruptedIOException
meth public int hashCode()
meth public java.lang.String toString()

CLSS public x.A
cons public <init>()
meth public void abc()
meth public void bar()
supr java.lang.Object
```
Example A–5  Contents of ./x3.sig

#Signature file v4.0
#Version

CLSS public java.lang.Object
cons public <init>()
meth protected java.lang.Object clone() throws java.lang.CloneNotSupportedException
meth protected void finalize() throws java.lang.Throwable
meth public boolean equals(java.lang.Object)
meth public final java.lang.Class<?> getClass()
meth public final void notify()
meth public final void notifyAll()
meth public final void wait() throws java.lang.InterruptedException
meth public final void wait(long) throws java.lang.InterruptedException
meth public final void wait(long, int) throws java.lang.InterruptedException
meth public int hashCode()
meth public java.lang.String toString()

CLSS public x.A
cons public <init>()
meth public static void abc()
supr java.lang.Object

Example A–6  Contents of ./x1+x2.sig

#Signature file v4.0
#Version

CLSS public java.lang.Object
cons public <init>()
meth protected java.lang.Object clone() throws java.lang.CloneNotSupportedException
meth protected void finalize() throws java.lang.Throwable
meth public boolean equals(java.lang.Object)
meth public final java.lang.Class<?> getClass()
meth public final void notify()
meth public final void notifyAll()
meth public final void wait() throws java.lang.InterruptedException
meth public final void wait(long) throws java.lang.InterruptedException
meth public final void wait(long, int) throws java.lang.InterruptedException
meth public int hashCode()
meth public java.lang.String toString()

CLSS public x.A
cons public <init>()
meth public void abc()
meth public void bar()
meth public void foo()
supr java.lang.Object
This section contains step-by-step examples that show how to produce test coverage reports using the API Coverage tool. The examples are located in the following directory:

\texttt{SIGTEST\_HOME/examples/userguide/apicover}

The appendix consists of the following examples. The examples build upon each other.

- Set Up the API Coverage Tool Ant Build Script
- Build API Coverage Tool Examples
- Run the Example Test Suite
- Run the Example Test Suite
- Use the API Coverage Tool to Calculate Test Coverage

\underline{Note:}  The examples use UNIX operating system syntax.

\underline{Caution:}  Code examples are an optional part of the SigTest distribution and are not required to use the SigTest tools. You can remove the example directory (\texttt{SIGTEST\_HOME/examples}) if you are concerned about security issues the examples might create.

### B.1 Example Procedures

The following procedures describes how to set up and use the API Coverage tool.

#### B.1.1 Set Up the API Coverage Tool Ant Build Script

1. Make \texttt{SIGTEST\_HOME/examples/userguide/apicover} the current directory.
2. Make sure that the \texttt{JAVA\_HOME} environment variable refers to JDK 5.0 or higher.
3. Edit the following file:

\texttt{SIGTEST\_HOME/examples/userguide/apicover/build.properties}

Set the \texttt{junit.jar} property to point to the \texttt{junit.jar} file. Obtain this file from version 3.7 or higher of the JUnit test harness product.
4. Make sure Apache Ant 1.7 or higher is available and properly configured on your system.
B.1.2 Build API Coverage Tool Examples

1. Run the ant command and specify the “build” target.
   This command builds the following:
   - API classes
   - Example application (helloWorld.jar)
   - Test suite
   - API documentation (doc/ directory)

2. Run the example application to verify that it built correctly.
   ```
   %java -jar helloWorld.jar
   ```
   The application’s output should look like this:
   
   Good morning, Brazil!
   Good morning, USA!
   Good evening, China!
   Good afternoon, Czech Republic!
   Good afternoon, Russia!
   Good afternoon, United Kingdom!
   Good evening, India!
   Good afternoon, Mali!
   Hello, world!

B.1.3 Run the Example Test Suite

1. Use the Ant “test” target to run the test suite.
   ```
   %ant test
   ```
   The test suite runs and produces the following output:
   
   [junit] Running TestGreet
   [junit] Tests run: 4, Failures: 0, Errors: 1, Time elapsed: 0.219 sec
   [junit] Test TestGreet FAILED

B.1.4 Generate a Signature File for the Tested API

1. Use the Ant “sigtest” target to generate the signature file.
   ```
   %ant sigtest
   ```
   The “sigtest” target executes the following command:
   
   java -jar sigtestdev.jar setup -filename reports/greetapi.sig -package
   com.sun.tdk.samples.helloworld.api -classpath helloWorld.jar:$JAVA_HOME/jre/lib/rt.jar

   This command generates the file reports/greetapi.sig. While the command runs
   you should see the following output:
   
   sigtest:
   [java] Class path: "helloWorld.jar:/opt/java/jdk1.5.0_16/jre/lib/rt.jar"
   [java] Constant checking: on
   [java] Found in total: 13217 classes
   [java] Selected by -Package: 5 classes
   [java] Written to sigfile: 8 classes
   [java] STATUS:Passed.
   [java] Java Result: 95
**B.1.5 Use the API Coverage Tool to Calculate Test Coverage**

1. Use the Ant “apicov” target to calculate static test coverage

```
%ant apicov
```

**B.1.6 Worst Case Mode**

The script executes the following command:

```
java -jar apicover.jar -ts test_classes -api reports/greetapi.sig -apiinclude com.sun.tdk.samples.helloworld.api -mode w -report reports/cov-worst-case.txt
```

This mode reports that about 37% of the API is covered by the tests. Some members that are defined in superclasses, such as `finalize()` or `clone()`, are marked as not covered. Some fields are also marked as not covered by the tests. The output written to a file named `reports/cov-worst-case.txt` and should appear as shown below.

Coverage Report

```
==========================================================================
Package          classes     members tested  %
Class            Member
==========================================================================
com               4          33       14       42%
--------------------------------------------------------------------------
com.sun             4          33       14       42%
--------------------------------------------------------------------------
com.sun.tdk          4          33       14       42%
--------------------------------------------------------------------------
com.sun.tdk.samples    4          33       14       42%
--------------------------------------------------------------------------
com.sun.tdk.samples.helloworld  4          33       14       42%
--------------------------------------------------------------------------
com.sun.tdk.samples.helloworld.api  4          33       14       42%
Greet               3          3          100%
GreetFactory         6          1          16%
    - clone()
    - equals(Object)
    - finalize()
    - hashCode()
    - toString()
Places               15         6        40%
    - getCountry()
    - toString()
    - valueOf(Class,String)
    - valueOf(String)
    - China
    - Czech
    - India
    - Mali
    - UK
TimeOfDay            9          4        44%
    - valuOf(Class,String)
    - AFTERNOON
    - EVENING
```
- MORNING
- NIGHT

Overall coverage 4 33 14 42%

Configuration
	ts  test_classes
	api  reports/greetapi.sig
	mode w

B.1.7 Real World Mode

The script executes the following command:

```
java -jar apicover.jar -ts test_classes -api reports/greetapi.sig -apiinclude com.sun.tdk.samples.helloworld.api -mode r -report reports/cov-realworld.txt
```

This estimation does not take into account overridden members defined in superclasses. Overall coverage is 52%.

The report is named reports/cov-realworld.txt and is shown below.

Coverage Report

<table>
<thead>
<tr>
<th>Package</th>
<th>classes</th>
<th>members tested</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>com</td>
<td>4</td>
<td>25</td>
<td>14</td>
</tr>
<tr>
<td>com.sun</td>
<td>4</td>
<td>25</td>
<td>14</td>
</tr>
<tr>
<td>com.sun.tdk</td>
<td>4</td>
<td>25</td>
<td>14</td>
</tr>
<tr>
<td>com.sun.tdk.samples</td>
<td>4</td>
<td>25</td>
<td>14</td>
</tr>
<tr>
<td>com.sun.tdk.samples.helloworld</td>
<td>4</td>
<td>25</td>
<td>14</td>
</tr>
<tr>
<td>com.sun.tdk.samples.helloworld.api</td>
<td>4</td>
<td>25</td>
<td>14</td>
</tr>
<tr>
<td>Greet</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>GreetFactory</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Places</td>
<td>13</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>- getCountry()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- valueOf(String)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Brazil</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- China</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Czech</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- India</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Mali</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- UK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TimeOfDay</td>
<td>8</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>- AFTERNOON</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- EVENING</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- MORNING</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
- NIGHT

| Overall coverage | 4 | 25 | 14 | 56% |

Configuration
- ts  test_classes
- api  reports/greetapi.sig
- mode  r

B.1.8 Real World Mode Without Fields and Enum Constants, detail Level 5

The script executes the following command:

```bash
```

In this case test coverage is 84%.

The report is named `reports/cov-realworld-without-fields.txt` and is shown below.

Coverage Report

<table>
<thead>
<tr>
<th>Package</th>
<th>classes</th>
<th>members tested</th>
<th>% count</th>
</tr>
</thead>
<tbody>
<tr>
<td>com</td>
<td>4</td>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>com.sun</td>
<td>4</td>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>com.sun.tdk</td>
<td>4</td>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>com.sun.tdk.samples</td>
<td>4</td>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>com.sun.tdk.samples.helloworld</td>
<td>4</td>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>com.sun.tdk.samples.helloworld.api</td>
<td>4</td>
<td>13</td>
<td>11</td>
</tr>
</tbody>
</table>

- Greet
  - greetPlaces()
  - greetWorld()
  - setWriter(java.io.Writer)

- GreetFactory
  - getInstance()

- Places
  - getCountry()
  - getTz()
  - shuffledValues()
  - valueOf(String)
  - values()

- TimeOfDay
  - getCurrentTime(java.util.TimeZone)
  - toString()
  - valueOf(String)
  - values()
Overall coverage                        4         13      11      84%
==========================================================================
Legend
+ covered member
- uncovered member
Configuration
   ts  test_classes
   api  reports/greetapi.sig
   excludeFields yes
   mode  r
This appendix describes the rules used by the Signature Test tool API migration feature.

## C.1 The Compatibility Rules

The following sections and tables describe the rules and show whether they break only source compatibility or both source and binary compatibility. Comments and clarifications follow the table for rules that have an asterisk after their number.

### Note:

A class is called *subclassable* if it has subclasses outside its package. In other words, this class is not final and has public constructor.

### C.1.1 General Rules

The rules from this group are triggered if the more specific rules from the other groups below are not triggered. For example, adding an interface method defined in rule 2.1, adding constructor in the rule 5.5. In such cases more general rule 1.1 is ignored.

<table>
<thead>
<tr>
<th>Rule #</th>
<th>Description</th>
<th>Breaks</th>
<th>Severity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 *</td>
<td>API (public or protected) type (class, interface, enum, annotation type) or member (method, field) added</td>
<td>Source</td>
<td>Warning</td>
</tr>
<tr>
<td>1.2</td>
<td>API (public or protected) type (class, interface, enum, annotation type) or member (method, field) removed</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>1.3</td>
<td>Narrowing type or member visibility - from public to non-public, from protected to package or private</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>1.4 *</td>
<td>Generification of the public API type</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

### C.1.2 Interfaces and Annotation Types

The rules in this group apply to interfaces and annotation types.
The Compatibility Rules

### Table C–2 API Migration Compatibility Rules - Interfaces and Annotation Types

<table>
<thead>
<tr>
<th>Rule #</th>
<th>Description</th>
<th>Breaks</th>
<th>Severity</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Add abstract methods</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>2.2</td>
<td>Add fields</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>2.3 *</td>
<td>Expand the set of superinterfaces (direct or indirect) if the added interface has a field (constant)</td>
<td>Source</td>
<td>Warning</td>
</tr>
<tr>
<td>2.4</td>
<td>Contract superinterface set (direct or inherited)</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>2.5 *</td>
<td>Add member without default value to annotation type</td>
<td>Source</td>
<td>Severe</td>
</tr>
<tr>
<td>2.6 *</td>
<td>Add member with default value to annotation type</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>2.7</td>
<td>Remove member from annotation type</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>2.8</td>
<td>Remove default value from member of annotation type</td>
<td>Both</td>
<td>Severe</td>
</tr>
</tbody>
</table>

**Note:** Some rules in this group apply only for class methods.

### Table C–3 API Migration Compatibility Rules - Interfaces and Class Methods

<table>
<thead>
<tr>
<th>Rule #</th>
<th>Description</th>
<th>Breaks</th>
<th>Severity</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Change signature and/or return type</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>3.2</td>
<td>Change last parameter from array type T[] to variable array T...</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Change last parameter from array T... to array type T[]</td>
<td>Source</td>
<td>Severe</td>
</tr>
<tr>
<td>3.4</td>
<td>Change normalized throw list</td>
<td>Source</td>
<td>Severe</td>
</tr>
<tr>
<td>3.5</td>
<td>Decrease access from public to protected</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>3.6 *</td>
<td>Increase access from protected to public if the class is subclassable</td>
<td>Source</td>
<td>Warning</td>
</tr>
<tr>
<td>3.7</td>
<td>Change method from abstract to non-abstract</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>3.8</td>
<td>Change method from non-abstract to abstract (if the class can be subclassed)</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>3.9</td>
<td>Change method from final to non-final</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>3.10</td>
<td>Change method from non-final to final</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>3.11</td>
<td>Change method from static to non-static</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>3.12</td>
<td>Change method from non-static to static</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>3.13</td>
<td>Change method from native to non-native</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>3.14</td>
<td>Change method from non-native to native</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>3.15</td>
<td>Change method from synchronized to non-synchronized</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>3.16</td>
<td>Change method from non-synchronized to synchronized</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

### C.1.3 Interfaces and Class Methods

The rules in this group apply to interfaces and class methods.

**Note:** Some rules in this group apply only for class methods.

### Table C–3 API Migration Compatibility Rules - Interfaces and Class Methods

<table>
<thead>
<tr>
<th>Rule #</th>
<th>Description</th>
<th>Breaks</th>
<th>Severity</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Change signature and/or return type</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>3.2</td>
<td>Change last parameter from array type T[] to variable array T...</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>Change last parameter from array T... to array type T[]</td>
<td>Source</td>
<td>Severe</td>
</tr>
<tr>
<td>3.4</td>
<td>Change normalized throw list</td>
<td>Source</td>
<td>Severe</td>
</tr>
<tr>
<td>3.5</td>
<td>Decrease access from public to protected</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>3.6 *</td>
<td>Increase access from protected to public if the class is subclassable</td>
<td>Source</td>
<td>Warning</td>
</tr>
<tr>
<td>3.7</td>
<td>Change method from abstract to non-abstract</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>3.8</td>
<td>Change method from non-abstract to abstract (if the class can be subclassed)</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>3.9</td>
<td>Change method from final to non-final</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>3.10</td>
<td>Change method from non-final to final</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>3.11</td>
<td>Change method from static to non-static</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>3.12</td>
<td>Change method from non-static to static</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>3.13</td>
<td>Change method from native to non-native</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>3.14</td>
<td>Change method from non-native to native</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>3.15</td>
<td>Change method from synchronized to non-synchronized</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>3.16</td>
<td>Change method from non-synchronized to synchronized</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

### C.1.4 Interfaces and Class Fields

The rules in this group apply to interfaces and class methods.

**Note:** Some rules from this group apply only for class fields.
C.1.5 Classes

The rules in this group apply to classes.

### Table C–4  API Migration Compatibility Rules - Interfaces and Class Fields

<table>
<thead>
<tr>
<th>Rule #</th>
<th>Description</th>
<th>Breaks</th>
<th>Severity</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Change type</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>4.2</td>
<td>Change/Remove constant value</td>
<td>Both</td>
<td>Warning</td>
</tr>
<tr>
<td>4.3</td>
<td>Decrease access</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>4.4</td>
<td>Increase access</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>4.5</td>
<td>Change from final to non-final</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>4.6</td>
<td>Change from non-final to final</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>4.7</td>
<td>Change from static to non-static</td>
<td>Bone</td>
<td>Severe</td>
</tr>
<tr>
<td>4.8</td>
<td>Change from non-static to static</td>
<td>Both</td>
<td>Severe</td>
</tr>
</tbody>
</table>

### Table C–5  API Migration Compatibility Rules - General

<table>
<thead>
<tr>
<th>Rule #</th>
<th>Description</th>
<th>Breaks</th>
<th>Severity</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Add non-abstract and non-static methods</td>
<td>Both</td>
<td>Warning</td>
</tr>
<tr>
<td>5.2</td>
<td>Add abstract methods (if the class can be subclassed)</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>5.3</td>
<td>Add static methods (if the class can be subclassed)</td>
<td>Both</td>
<td>Warning</td>
</tr>
<tr>
<td>5.4</td>
<td>Remove constructors</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>5.5</td>
<td>Add first constructor with arguments or throws clause</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>5.6</td>
<td>Add fields</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>5.7</td>
<td>Expand implemented interface set (direct or indirect)</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>5.7.1</td>
<td>The added interface adds abstract methods</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>5.7.2</td>
<td>The new interface adds fields or inner classes</td>
<td>Source</td>
<td>Severe</td>
</tr>
<tr>
<td>5.7.3</td>
<td>If 5.7.1 and 5.7.3 are not true</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>5.8</td>
<td>Contract implemented interface set (direct or indirect)</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>5.9</td>
<td>Expand superclass set (direct or indirect)</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>5.9.1</td>
<td>Add superclass adds abstract method (see rules 5.1 - 5.3)</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>5.9.2</td>
<td>Add superclass adds field (see rule 5.6)</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>5.9.3</td>
<td>Other cases</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>5.10</td>
<td>Contract superclass set (direct or indirect)</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>5.11</td>
<td>Change abstract to non-abstract</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>5.12</td>
<td>Change non-abstract to abstract (if the class can be subclassed)</td>
<td>Both</td>
<td>Severe</td>
</tr>
<tr>
<td>5.13</td>
<td>Change final to non-final</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>5.14</td>
<td>Change non-final to final</td>
<td>Both</td>
<td>Severe</td>
</tr>
</tbody>
</table>

C.2 Comments and Clarifications

The following sections contain comments and clarifications that make the rules more clear.
C.2.1 Rule 1.1

Adding a class can theoretically break source code compatibility because new classes can be incorrectly resolved in an existing client’s code with type-import-on-demand declarations (also known as wildcard imports). This can happen if the code uses another type with the same simple name. For example:

```java
//client’s code
import com.acme.*;
import com.client.*;
.....
Policy p = new Policy();   // this is com.client.Policy
.....
```

In this case, after adding class `com.acme.Policy`, the compiler raises the error `reference to Policy is ambiguous, both class com.acme.Policy in com.acme and class com.client.Policy in com.client match.`

This rule is considered a warning because it will probably not affect binary compatibility and adhering to this rule makes API evolution very difficult.

C.2.2 Rule 1.4

Generics are a facility of generic programming that was added to the Java programming language as part of Java SE version 5.0. Generics allow a type or method to operate on objects of various types while providing compile-time type safety. Generification upgrades types using support-to-be-specified-later types that are instantiated as needed for specific types that are provided as type parameters.

The Java programming language implements generics using erasure, which ensures that legacy and generic versions usually generate identical class files, except for some auxiliary information about types. Binary compatibility is not broken because it is possible to replace a legacy class file with a generic class file without changing or recompiling any client code.

To facilitate interfacing with non-generic legacy code, it is also possible to use the erasure of a parameterized type as a type. Such a type is called a raw type (*Java Language Specification* 3/4.8). Allowing the raw type also ensures backward compatibility for source code.

According to this, the following versions of the `java.util.Iterator` class are both binary and source code backward compatible:

- **Class** `java.util.Iterator` as it is defined in Java SE version 1.4:

```java
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}
```

- **Class** `java.util.Iterator` as it is defined in Java SE version 5.0:

```java
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```
C.2.3 Rule 2.3

Adding a superinterface with a constant field can shadow another entity with the same simple name. Consider the following code:

```java
public interface Poet{
    boolean LITERATE = true;
}

public interface Playwright{
    boolean LITERATE = true;
}

public interface Shakespeare extends Poet {
}

Suppose that a new version of the Shakespeare interface implements interface Playwright as well as Poet as shown here:

```java
public interface Shakespeare extends Poet, Playwright {
}
```

The following client code will not compile because the reference to `LITERATE` is ambiguous.

```java
// client code
public class ShakespeareImpl implements Shakespeare {
    void introduce() {
        System.out.println("Hi, my name is Shakespeare and I'm " + LITERATE ?
            "quite literate" : "rather illiterate ");
    }
}
```

C.2.4 Rules 2.5, 2.6

The example below shows how adding a member without a default value to an annotation type breaks source code compatibility.

```java
// annotation type v1
@interface Agent{
    String name();
}

// client's code which uses this annotation type v1
@Agent(name="James Bond")

// annotation type v2.1
@interface Agent{
    String name();
    String id(); // added member
}

// legacy code is not compilable due to
// error - annotation Agent is missing id
@Agent(name="James Bond")
```
C.2.5 Rule 3.6
Changing a method from protected to public can break source code compatibility if
this method was overridden as protected. In this case the legacy code can not be
recompiled because “access narrowing” is prohibited in the Java programming
language. This rule is only a warning because it does not affect binary compatibility,
and the probability of its affecting source code compatibility is very low.

C.2.6 Rule 4.2
Changing or removing constant values can break source code compatibility. For
example, consider the following client code example. An integer constant named
NOTHING with a value of 0 is used:

```java
switch (i) {
    case NOTHING:
        // some actions
    case -1:
        // some other actions
}
```

Assume that the value of the constant NOTHING is changed from 0 to -1. The client code
will not compile because of the duplicate case label. This rule is only a warning
because it does not affect binary compatibility, and the probability of its affecting
source code compatibility is very low.

C.2.7 Rule 5.1
Adding a regular method to a subclassable class can break source code and binary
compatibility because a subclass can have a method with the same signature but with
weaker access privileges. For example consider the following code example:

```java
class ClientClass extends APIClass {
    private void foo() {} 
}
```

Assume that the class APIClass is changed, and the method

```java
protected void foo() {} 
```

is added. The code cannot be recompiled due of an error that generates the following
error message:

```
foo() in ClientClass cannot override foo() in APIClass; attempting to assign
weaker access privileges; was protected
```

Binary compatibility is broken as defined in JLS 3/3.14.12
C.2.8 Rule 5.3

Adding a static method to a subclassable class can break source code and binary compatibility, because a subclass can have a method with the same signature. For example consider the following code example:

```java
class ClientClass extends APIClass {
    protected void foo() {} // }
}
```

Suppose that the class `APIClass` is changed and the method

```java
protected static void foo() {} // }
```

is added. The code cannot be recompiled due to an error that generates the following error message:

```
foo() in ClientClass cannot override foo() in APIClass; overridden method is static
```

Binary compatibility is be broken as defined in JLS 3/3.14.12

C.2.9 Rule 5.7.2

As in the case of rule 2.3, adding a superinterface with a constant field can shadow another entity with the same simple name.
Merge Command Operative Principles

This appendix describes the command operative principles used by the Signature Test tool.

D.1 Merge Command Operative Principles

The Merge command operates according to the following principles, where A and B are input APIs that are combined into the resulting API C:

- The Merge operation is commutative, so with API A and B, $A + B = B + A$.
- It recognizes either binary or source compatibility when merging APIs.
- For any application $X$ that is compatible with either API $A$ or $B$, when $A$ and $B$ are merged then $X$ must be compatible with the resulting API $C$.
- The resulting API $C$ cannot contain a class that is not found in either of the $A$ or $B$ input APIs. This means that any class in $C$ has to have corresponding classes in either $A$ or $B$ or both $A$ and $B$.
- API $C$ must not contain a class member that is not found in its corresponding classes in $A$ and $B$. This applies only to declared class members and not inherited members.
- If some class in $A$ or $B$, or both, has a member that overrides a member from a superclass, then the corresponding class in $C$ must also have this overriding member.
- Each API element in $C$ has a set of attributes derived from the attributes of its corresponding elements in $A$ and $B$, and this is the smallest possible set of attributes that does not break compatibility. So if $\text{attr}$ is an attribute of an element from API $C$, then $\text{attr}$ must be defined for the corresponding element from $A$ or $B$, and $\text{attr}$ cannot be omitted without breaking compatibility between $A$ and $C$ or between $B$ and $C$.
- No unnecessary APIs or relationships between classes or interfaces can be introduced.

The basic algorithmic rules for combining two input APIs $A$ and $B$ into a signature file that represents the resulting API $C$ are as follows:

- If one of the input APIs $A$ or $B$ contains an element that the other does not, then this element goes into the resulting signature file of API $C$ without modification except for the following case: If the element in question is the first declared class member in the inheritance chain of input API $A$ or $B$, and the other input API inherits the same element, then this element represented the resulting API $C$. 

If both of the input APIs contain two identical elements, only one of them is represented in the resulting API.

If both of the input APIs contain a corresponding element, but with a different set of attributes, then either of the following occurs:

- A conflict wherein the resulting API cannot exist.
- A compromise wherein the new element with a composite set of attributes is created and it is represented in the resulting API-set.

D.1.1 Element Handling by Merge

General rules for handling elements of all kinds during the Merge process are as follows.

- When there are two different access modifiers select the more visible one.
  For example, if A is a public int foo, and B is protected int foo, then the merge into C results into public int foo.

- If the elements differ in the final modifier, do not include it. If a class is final, then all of its methods are implicitly final according to Section 8.4.3.3 of The Java Language Specification, 2nd Edition.

- If corresponding elements differ in the static modifier, then declare a conflict.

Element-specific rules are as follows:

- If corresponding classes differ in the abstract modifier, then declare a conflict.

- Apply the following rules for classes or interfaces and nested classes or interfaces, where for the purpose of this description, c1 and c2 are corresponding classes from the input APIs:
  If a superclass of c1 is a subclass of a superclass of c2, use the superclass of c1 as the superclass for the new element. Otherwise, if a superclass of c2 is a subclass of a superclass of c1, use the superclass of c2 as the superclass for the new element. If neither of the previous two conditions are possible, then declare a conflict.

- For classes or interfaces and nested classes or interfaces, create a combined set of superinterfaces of the corresponding classes and dismiss duplicates. Use the combined set for the new element.

- For methods and constructors, construct a throws list as follows:
  - In binary compatibility mode, an empty throws list results independently of the source lists.
  - In source compatibility mode, both throws lists are normalized as described in Table 4–3 before they are compared. If the normalized lists are equal, one is used as the result, otherwise, a conflict is declared.

- Methods that differ in the abstract modifier are not included.

- If a method result type is not identical a conflict is declared.

- If a field value type is not identical a conflict is declared.

- If a field element differs in the volatile modifier, it is included.

- Process constant field values as follows:
  - If one of the fields has a constant value and other does not, include the constant value in the result field.
If both fields have a constant value then declare a conflict if the values are different, otherwise include the value in the result field.
This appendix describes how to use the SigTest tools in Ant build scripts.

E.1 Using the Setup Command From an Ant Script

The Signature Test tool Setup command can be invoked from an Ant script. The sigtestdev.jar file contains the corresponding Ant task.

Use the following task definition in your Ant build script:

```xml
<taskdef name="setup" classname="com.sun.tdk.signaturetest.ant.ASetup" classpath="sigtestdev.jar"/>
```

Table E–1 lists required and optional attributes and nested elements for the “setup” task.

**Table E–1 Setup Attributes Available for Ant Scripts**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Required</strong></td>
<td></td>
</tr>
<tr>
<td>“package” attribute or nested “package” elements</td>
<td>Corresponds to the -package option</td>
</tr>
<tr>
<td>“classpath” attribute or nested “classpath” elements</td>
<td>Corresponds to the -classpath option</td>
</tr>
<tr>
<td>“filename” attribute</td>
<td>Corresponds to the -filename option</td>
</tr>
<tr>
<td><strong>Optional:</strong></td>
<td></td>
</tr>
<tr>
<td>“failonerror” attribute</td>
<td>Stops the build process if the command exits with an error. Default value is false</td>
</tr>
<tr>
<td>“apiVersion” attribute</td>
<td>Corresponds to -apiVersion. Set API version for signature file</td>
</tr>
<tr>
<td>“exclude” attribute or nested “exclude” elements</td>
<td>Corresponds to -exclude option. Specifies package(s) or class(es), which is not required to be tested</td>
</tr>
<tr>
<td>“negative” attribute</td>
<td>Treats a failed result as successful and a successful result as failed. Used for negative tests. Default value is “false”.</td>
</tr>
</tbody>
</table>

The following code shows how to use the Setup command in an Ant script:

```xml
<target name="td" description="Setup task definition">
```
E.2 Using the SignatureTest Command From an Ant Script

The SignatureTest command can be invoked from Ant script. Both sigtestdev.jar and sigtest.jar contain corresponding Ant tasks. Unlike the ordinary SignatureTest command, its Ant wrapper can run the test only in static mode—reflection mode is not supported. Use the following task definition in your Ant build script:

```xml
<taskdef name="test" classname="com.sun.tdk.signaturetest.ant.ATest" classpath="sigtest.jar"/>
```

Table E–2 lists required and optional attributes and nested elements for “test” Ant task.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required:</td>
<td></td>
</tr>
<tr>
<td>“package” attribute or nested “package” elements</td>
<td>Corresponds to the -package option</td>
</tr>
<tr>
<td>“classpath” attribute or nested “classpath” elements</td>
<td>Corresponds to the -classpath option</td>
</tr>
<tr>
<td>“filename” attribute</td>
<td>Corresponds to the -filename option</td>
</tr>
<tr>
<td>Optional:</td>
<td></td>
</tr>
<tr>
<td>“failonerror” attribute</td>
<td>Stops the build process if the command exits with an error. Default is false</td>
</tr>
<tr>
<td>“apiVersion” attribute</td>
<td>Corresponds to the -apiVersion option. Sets the API version for signature files.</td>
</tr>
<tr>
<td>“exclude” attribute or nested “exclude” elements</td>
<td>Corresponds to the -exclude option. Specifies package(s) or class(es), do not require testing.</td>
</tr>
<tr>
<td>“binary” attribute</td>
<td>Corresponds to the “-mode bin” option. Runs the test in binary mode. Default is “false”.</td>
</tr>
<tr>
<td>“errorAll” attribute</td>
<td>Corresponds to the -errorAll option. Handles warnings as errors. Default is “false”.</td>
</tr>
<tr>
<td>“debug” attribute</td>
<td>Corresponds to the -debug option. Prints debug information such as detailed stack traces. Default is “false”.</td>
</tr>
</tbody>
</table>
The following code sample shows how to use the SignatureTest command with an Ant script.

```xml
<target name="td" description="Signature test task definition">
  <taskdef name="test" classname="com.sun.tdk.signaturetest.ant.ATest"
           classpath="${sigtest.home}/sigtest.jar"/>
</target>

<target name="test" description="Runs migration compatibility test for com.acme.openapi package with subpackages, tests v2 against v1 signature file"
        depends="td">
  <test failonerror="true" apiVersion="openapi-v2"
        filename="acme-openapi-v1.sig" backward="true" output="st_report.txt">
    <package name="com.acme.openapi"/>
    <exclude class="com.acme.openapi.NotTested"/>
    <classpath>
      <pathelement location="${java.home}/jre/lib/rt.jar"/>
      <pathelement location="${testd.home}/lib/acmeAPIv2.jar"/>
    </classpath>
  </test>
</target>
```

The following code sample shows how to use the SignatureTest command with an Ant script.

```xml
<target name='td' description='Signature test task definition'>
  <taskdef name='test' classnames='com.sun.tdk.signaturetest.ant.ATest'
           classpath='${sigtest.home}/sigtest.jar'/>
</target>

<target name='test' description='Runs migration compatibility test for com.acme.openapi package with subpackages, tests v2 against v1 signature file'
        depends='td'>
  <test failonerror='true' apiVersion='openapi-v2'
        filename='acme-openapi-v1.sig' backward='true' output='st_report.txt'>
    <package name='com.acme.openapi' />
    <exclude class='com.acme.openapi.NotTested'/>
    <classpath>
      <pathelement location='${java.home}/jre/lib/rt.jar'/>
      <pathelement location='${testd.home}/lib/acmeAPIv2.jar'/>
    </classpath>
  </test>
</target>
```

### E.3 Using the Merge Command From an Ant Script

The Signature Test tool `Merge` command can be invoked from an Ant script. Both the `sigtestdev.jar` and `sigtest.jar` files contain the corresponding Ant task. Use the following task definition in your Ant build script:

```xml
<taskdef name='merge' classname='com.sun.tdk.signaturetest.ant.AMerge'
         classpath='sigtest.jar'/>
```

Table E–3 and Table E–4 list required and optional attributes and nested elements for the `merge` task.

#### Table E–3 Required Setup Attributes Available for Ant Scripts

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“file” nested elements</td>
<td>Corresponds to the <code>-files</code> option</td>
</tr>
<tr>
<td>“write” attribute</td>
<td>Corresponds to the <code>-write</code> option</td>
</tr>
</tbody>
</table>
The following code shows how to use the Merge command in an Ant script:

```xml
<target name="mergeFiles" description="Merges signature files">
  <taskdef name="merge" classname="com.sun.tdk.signaturetest.ant.AMerge" classpath="${sigtest.jar} />
  <merge write="${build.test.dir}/core_and_loaders.sig">
    <file name="${build.test.dir}/core.sig" />
    <file name="${build.test.dir}/loaders.sig" />
  </merge>
</target>
```

### E.4 Using APICheck From an Ant Script

The API Check tool command can be invoked from an Ant script. The apicheck.jar file contains the corresponding Ant task. Use the following task definition in your Ant build script:

```xml
<taskdef name="apicheck" classname="com.sun.tdk.apicheck.ant.ACheck" classpath="apicheck.jar:sigtest.jar" />
```

All required and optional attributes and nested elements for the “apicheck” task are the same as used for the Signature Test Tool “test” Ant command and are listed in Table E–2.

See B.1.1 for an example that shows how to set up APICheck in an Ant script.
Using the SigTest Tools With Apache Maven

The Apache Maven SigTest plug-in (sigtest-plugin) makes the functionality of the Sigtest, Setup and Merge commands available from Maven 2 build scripts. The Maven APICover plug-in (apicover-plugin) makes the functionality of the APICover tool available to Maven 2 build scripts. All goals are bound by default to the “test” life cycle phase.

F.1 Goal Overview

The four sigtest-plugin goals are described in Table F–1:

<table>
<thead>
<tr>
<th>Goal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sigtest-plugin:setup</td>
<td>Provides Setup command functionality to Maven 2 build scripts. Creates a signature file from an API defined by a specification or a reference implementation.</td>
</tr>
<tr>
<td>sigtest-plugin:test</td>
<td>Provides static mode SignatureTest functionality to Maven 2 build scripts. Compares the reference API represented in the signature file to the API under test and produces a report.</td>
</tr>
<tr>
<td>sigtest-plugin:merge</td>
<td>Provides Merge functionality to Maven 2 build scripts. Creates a combined signature file from several signature files representing different Java platform APIs in one Java runtime environment according to the JSR 68 rules.</td>
</tr>
<tr>
<td>apicover-plugin:apicover</td>
<td>Provides API Coverage tool functionality to Maven 2 build scripts. This goal is used to estimate the test coverage that a test suite under development affords to implementations of its related API specification.</td>
</tr>
</tbody>
</table>

F.2 Goal Descriptions

The following sections describe the requirements and parameters for each of the goals listed in Table F–1. An example follows each section.

F.2.1 sigtest-plugin:setup

The setup goal accepts a reference implementation of an API as input. The command processes API input in order to generate a signature file that represents the API to be used as a reference for comparison during signature testing.

- Full name: com.sun.tdk:sigtest-plugin:2.2:setup
- Requires a Maven 2.0 project to be executed.
Binds by default to the life cycle phase: test

**Table F–2** Required Parameters (*sigtest-plugin:setup*)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Corresponding Setup Command Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fileName</td>
<td>String</td>
<td>-filename</td>
<td>Specifies the name of the signature file to be created.</td>
</tr>
<tr>
<td>pathElements</td>
<td>List</td>
<td>-classpath</td>
<td>Specifies the path to one or more APIs that generate the signature file. Can contain multiple directories, Zip files or JAR files.</td>
</tr>
<tr>
<td>packages</td>
<td>List</td>
<td>-package</td>
<td>Specifies classes or packages to be included in the signature file including its subpackages if a package is specified.</td>
</tr>
</tbody>
</table>

**Table F–3** Optional Parameters (*sigtest-plugin:setup*)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Corresponding Setup Command Option</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>excludes</td>
<td>List</td>
<td>-exclude</td>
<td></td>
<td>Specifies a package or class to be excluded from the signature file, including its subpackages.</td>
</tr>
<tr>
<td>nonClosedFile</td>
<td>boolean</td>
<td>-NonClosedFile</td>
<td>FALSE</td>
<td>Causes direct and indirect superclasses and superinterfaces of tested classes not to be included in the signature file. This eliminates cases where they do not belong to the scope defined by “packages” parameters.</td>
</tr>
<tr>
<td>apiVersion</td>
<td>String</td>
<td>-apiVersion</td>
<td></td>
<td>Specifies the API version string to be recorded in the second line of the signature file.</td>
</tr>
<tr>
<td>failOnError</td>
<td>boolean</td>
<td></td>
<td>TRUE</td>
<td>Stops the build process if the command exits with an error.</td>
</tr>
<tr>
<td>negative</td>
<td>boolean</td>
<td></td>
<td>FALSE</td>
<td>Treats tests that do not succeed as having passed, and tests that do succeed as having failed. Used for negative tests.</td>
</tr>
</tbody>
</table>

**Example F–1** Sample Plugin Configuration (*pom.xml*)

```xml
<project>  
  . . .  
  <build>  
    . . .  
    <plugins>  
      <plugin>  
        . . .  
        <groupId>com.sun.tdk</groupId>  
        <artifactId>sigtest-plugin</artifactId>  
        <configuration>  
          . . .  
        </configuration>  
      </plugin>  
    </plugins>  
  </build>  
</project>
```
F.2.2 sigtest-plugin:test

The test goal compares the reference API (represented in a signature file) to the API under test and produces a report. Unlike the SignatureTest command it only supports static mode—reflection is not supported.

- Full name: com.sun.tdk:sigtest-plugin:2.2:test
- Requires a Maven 2.0 project to be executed.
- Binds by default to the life cycle phase: test

Table F–4 Required Parameters (sigtest-plugin:test)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Corresponding Setup Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fileName</td>
<td>String</td>
<td>-filename</td>
<td>Specifies the name of the signature file to be created.</td>
</tr>
<tr>
<td>pathElements</td>
<td>List</td>
<td>-classpath</td>
<td>Specifies the path to one or more APIs that generate the signature file. Can specify multiple directories, Zip files or JAR files.</td>
</tr>
<tr>
<td>packages</td>
<td>List</td>
<td>-package</td>
<td>Specifies classes or packages to be included in the signature file, including its subpackages if a package is specified.</td>
</tr>
</tbody>
</table>

Table F–5 Optional Parameters (sigtest-plugin:test)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Corresponding Signature Test Option</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>excludes</td>
<td>List</td>
<td>-exclude</td>
<td></td>
<td>Specifies a package or class to be excluded from the signature file, including its subpackages.</td>
</tr>
<tr>
<td>apiVersion</td>
<td>String</td>
<td>-apiVersion</td>
<td></td>
<td>Specifies the API version string to be recorded in the second line of the signature file.</td>
</tr>
</tbody>
</table>
Example F–2  Sample Plugin Configuration (pom.xml)

    <project>
      . . .
    <build>
      . . .
      <plugins>
        . . .
        <plugin>
          . . .
          <groupId>com.sun.tdk</groupId>
          <artifactId>sigtest-plugin</artifactId>
          <configuration>
            <fileName>java-awt-lang-math.sig</fileName>
            <packages>
              <package>java.awt</package>
              <package>java.lang</package>
              <package>java.math</package>
            </packages>
            <excludes>
              <exclude>java.lang.Math</exclude>
              <exclude>java.lang.ref</exclude>
            </excludes>
            <pathElements>
              <pathElement>${ref.jhome}/jre/lib/rt.jar</pathElement>
            </pathElements>
            <apiVersion>Java Six</apiVersion>
            <failOnError>false</failOnError>
            <human>true</human>
            <backward>true</backward>
            <out>report.txt</out>
            <debug>true</debug>
          </configuration>
        </plugin>
      </plugins>
    </build>
  </project>
F.2.3 sigtest-plugin:merge

The `merge` goal command combines (merges) a number of input signature files into a single signature file.

- **Full name:** com.sun.tdk:sigtest-plugin:2.2:merge
- **Requires a Maven 2.0 project to be executed.**
- **Binds by default to the life cycle phase: test**

### Table F–6 Required Parameters (sigtest-plugin:merge)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Corresponding Merge Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>write</td>
<td>String</td>
<td>-Write</td>
<td>Specifies the resulting output signature file.</td>
</tr>
<tr>
<td>files</td>
<td>List</td>
<td>-Files</td>
<td>Specifies the names of the signature files to be merged.</td>
</tr>
</tbody>
</table>

### Table F–7 Optional Parameters (sigtest-plugin:merge)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Corresponding Merge Option</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary</td>
<td>boolean</td>
<td>-Binary</td>
<td>FALSE</td>
<td>Specifies that binary mode be used during the signature test instead of the source code compatibility mode.</td>
</tr>
<tr>
<td>failOnError</td>
<td>boolean</td>
<td></td>
<td>TRUE</td>
<td>Stops the build process if the command exits with an error.</td>
</tr>
<tr>
<td>negative</td>
<td>boolean</td>
<td></td>
<td>FALSE</td>
<td>Treats tests that do not succeed as having passed, and tests that do succeed as having failed. Used for negative tests.</td>
</tr>
</tbody>
</table>

### Example F–3 Sample Plugin Configuration (pom.xml)

```xml
<project>
  
  <build>
    
    <plugins>
      
      <plugin>
        <groupId>com.sun.tdk</groupId>
        <artifactId>sigtest-plugin</artifactId>
        <configuration>
          <files>
            <file>baseAPI.sig</file>
            <file>optionalAPI1.sig</file>
          </files>
        </configuration>
      </plugin>
    
    </plugins>
    
  </build>

</project>
```
The apicover-plugin:apicover goal is used to estimate the test coverage that a test suite under development affords to implementations of its related API specification. It does this by determining how many public class members the test suite references within the API specification that it is designed to test against. The tool uses a signature file representation of the API specification as the basis of its specification analysis.

- Full name: com.sun.tdk:apicover-plugin:2.2:apicover
- Requires a Maven 2.0 project to be executed.
- Binds by default to the life cycle phase: test

### Table F–8 Required Parameters (apicover-plugin:apicover)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Corresponding APICover Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tests</td>
<td>List</td>
<td>-ts</td>
<td>Specifies the path to the test suite classes to be analyzed. Also accepts a JAR file.</td>
</tr>
<tr>
<td>api</td>
<td>String</td>
<td>-api</td>
<td>Specifies the location of the signature file representing the API under examination.</td>
</tr>
</tbody>
</table>

### Table F–9 Optional Parameters (apicover-plugin:apicover)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Corresponding APICover Option</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tsIncludes</td>
<td>List</td>
<td>-tsInclude</td>
<td></td>
<td>Recursively includes classes from the specified test suite package.</td>
</tr>
<tr>
<td>tsIncludeWs</td>
<td>List</td>
<td>-tsIncludeW</td>
<td></td>
<td>Includes classes from the specified test suite package without subpackages.</td>
</tr>
<tr>
<td>tsExcludes</td>
<td>List</td>
<td>-tsExclude</td>
<td></td>
<td>Recursively excludes classes from the specified test suite package.</td>
</tr>
<tr>
<td>apiIncludes</td>
<td>List</td>
<td>-apiInclude</td>
<td></td>
<td>Recursively includes classes from the specified API package.</td>
</tr>
<tr>
<td>apiIncludeWs</td>
<td>List</td>
<td>-apiIncludeW</td>
<td></td>
<td>Includes classes from the specified API package without subpackages.</td>
</tr>
</tbody>
</table>
**Table F–9 (Cont.) Optional Parameters (apicover-plugin:apicover)**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Corresponding APICover Option</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>apiExcludes</td>
<td>List</td>
<td>-apiExclude</td>
<td></td>
<td>Recursively excludes classes from the specified API package.</td>
</tr>
<tr>
<td>excludeLists</td>
<td>List</td>
<td>-excludeList</td>
<td></td>
<td>Specifies the file name of an exclude list that contains API elements to be excluded from the coverage calculation.</td>
</tr>
<tr>
<td>excludeInterfaces</td>
<td>boolean</td>
<td>-excludeInterface</td>
<td>FALSE</td>
<td>Excludes all interface classes.</td>
</tr>
<tr>
<td>excludeAbstractClasses</td>
<td>boolean</td>
<td>-excludeAbstractClasses</td>
<td>FALSE</td>
<td>Excludes all abstract classes including interfaces.</td>
</tr>
<tr>
<td>excludeAbstractMethods</td>
<td>boolean</td>
<td>-excludeAbstractMethods</td>
<td>FALSE</td>
<td>Excludes all abstract methods from all classes and interfaces.</td>
</tr>
<tr>
<td>excludeFields</td>
<td>boolean</td>
<td>-excludeFields</td>
<td>FALSE</td>
<td>Excludes all fields from all classes and interfaces.</td>
</tr>
<tr>
<td>includeConstantFields</td>
<td>boolean</td>
<td>-includeConstantFields</td>
<td>FALSE</td>
<td>Specifies the inclusion of all final fields of any type in the coverage calculation.</td>
</tr>
<tr>
<td>mode</td>
<td>String</td>
<td>-mode</td>
<td>w (worst case)</td>
<td>Specifies the mode of coverage calculation as “w” which represents the worst case or “r” which represents the real world.</td>
</tr>
<tr>
<td>detail</td>
<td>String</td>
<td>-detail</td>
<td>2</td>
<td>Specifies the level of report detail as an integer from 0-4</td>
</tr>
<tr>
<td>format</td>
<td>String</td>
<td>-format</td>
<td>plain</td>
<td>Specifies the report format as plain text or XML. The values can be “plain” or “xml”. Note that an example of the XML output is available in SIGTEST_HOME/msc.</td>
</tr>
<tr>
<td>report</td>
<td>String</td>
<td>-report</td>
<td>See the description.</td>
<td>Specifies the location of the generated report file. The path argument defaults to the working directory when only a file name is specified. If a file name is not specified, it defaults to the Maven logger.</td>
</tr>
<tr>
<td>out</td>
<td>String</td>
<td>-out</td>
<td>See the description</td>
<td>Generates a specially formatted report file for use by other applications and places it at the specified location/file name. The path defaults to the working directory.</td>
</tr>
</tbody>
</table>

**Example F–4  Sample Plugin Configuration (pom.xml)**

```xml
<project>
```
...<build>
...
<plugins>
...
<plugin>
...
  <groupId>com.sun.tdk</groupId>
  <artifactId>apicover-plugin</artifactId>
  <configuration>
    <api>publicAPIv3.sig</api>
    <tests>
      <test>${ts.path}/testSet1.jar</test>
      <test>${ts.path}/testSet2.jar</test>
      <test>${ts.path}/testSet3.jar</test>
    </tests>
    <apiIncludes>
      <apiInclude>com.acme.api</apiInclude>
    </apiIncludes>
    <mode>r</mode>
    <detail>4</detail>
    <format>plain</format>
    <report>${report.path}/covrep_real.txt</report>
  </configuration>
...
</plugin>
...
</plugins>
...
</build>
...
</project>
This appendix gives an example of creating a custom tool based on the SigTest tool and its APIs. The example tool checks two implementations of a set of Java classes for serialization compatibility. You can create similar tools based on this example.

This appendix contains the following topics:

- Section G.1, "Overview"
- Section G.2, "Usage"
- Section G.3, "Implementation"

G.1 Overview

The serialization tool is an example of using the SigTest APIs to create a new kind of tool. The serialization tool verifies that two implementations of one set of classes can successfully serialize and deserialize instances of each other. For a description of serialization, see http://docs.oracle.com/javase/7/docs/technotes/guides/serialization/index.html.

G.2 Usage

The serialization tool has two commands and is run much like the base SigTest tool.

Setup command:
$ java com.sun.tdk.sertest.SerSetup [options]

Test command:
$ java com.sun.tdk.sertest.SerTest [options]

Be certain that sigtestdev.jar and sertest.jar are in the classpath. Most SigTest tool options are valid for the serialization tool.

The serialization tool differs from the base SigTest tool in the following ways:

- Only java.io.Serializable classes are taken (except enums).
- For all classes, only non-transient and non-static fields are tracked. Also serialVersionUID is required and its value is tracked.
- Any serializable class must have a serialVersionUID.
- Because of reflection limitations, access to class members via reflection is not supported, and the test command is always static regardless of the -static option.
- The setup command always generates a -nonclosed signature file.
- Signature files generated by this tool’s setup command are incompatible with, and must not be used by, the base SigTest tool or any other custom tool derived from it. Conversely, signature files generated by other tools must not be used with the serialization tool.

G.3 Implementation

The serialization tool is implemented in the `com.sun.tdk.sertest` package in `SigTest-Directory/examples/sertest/`. There are three classes:

**SerSetup**
Main class for setup command

**SerTest**
Main class for test command

**SerUtil**
Common code for setup and test commands

For implementation details, see the Javadoc in the `SigTest-Directory/javadoc` directory.