JCov FAQ

This page contains recommendations on solving various problems related to JCov

JCov OS Community

What is JCov?

JCov is a ***J***ava code ***Cov***erage tool which provides a means to measure and analyze dynamic code coverage of Java programs. "Dynamic" means that code coverage data is collected when a program executes, unlike static code coverage which is collected without running the program. JCov provides functionality to collect method, linear block and branch coverage. It is also able to show a program's source code annotated with coverage information.

JCov helps to estimate the level of source code coverage of a Java application under test and determine untested code - parts of application not executed by tests.

What are the licensing terms for OS JCov?

The JCov project is licensed under the GNU Public License, version 2, with the Classpath Exception. See the license page for more information.

What is the classpath exception?

The classpath exception was developed by the Free Software Foundation's GNU/Classpath Project (see http://www.gnu.org/software/classpath/license.html). It allows you to link an application available under any license to a library that is part of software licensed under GPL v2, without that application being subject to the GPL's requirement to be itself offered to the public under the GPL.

Why do you need the classpath exception?

If JCov was distributed under GPL v2, that application could be subject to the requirements of the GPL that all code that is shipped as part of a "work based on the [GPL] program" also be GPL licensed. Accordingly, a GPL license exception is needed that specifically excludes from this licensing requirement any application that links to the GPL implementation. The classpath exception accomplishes this.

What is the relationship between the JCov project and the OpenJDK - CodeTools community?

The OpenJDK - CodeTools project is the Java open source home for technologies used to test a Java Platform.

Similar to the JT harness project, the JCov project started out as a commercially released test technology, and has been released to the Java open source community. The JCov project also shares the same governance model and licensing requirements as the OpenJDK CodeTools Project.

How can I submit or suggest changes to the JCov project?

We encourage you to contribute source code to the JCov project. In accordance with the OpenJDK project governance process, you can submit code to the project leader either as a user, a developer, or as a committer. If your code is accepted to be integrated into the source tree, you will be asked to sign a Contributor's Agreement similar to what Apache Software Foundation requires. For more information about the governance process and the various roles for members in the community, please review OpenJDK Governance.

What does the JCov2.0 tag mean in the JCov repository

JCov tool versions are typically aligned with JDK developments. Since JCov is used to evaluate code coverage data for the JDK development process, it is convenient for development tagging of feature sets in the JCov OpenJDK repo. Specifically, the JCov2.0 release is aligned with the release of JDK8.

What can we expect in next JCov version?

The next JCov milestone will be connected with JDK9. Specifically, this means that JCov will support changes in JDK infrastructure - as proposed for project Jigsaw (http://openjdk.java.net/projects/jigsaw/). Other future JCov directions include: more dynamic and interactive reports, coverage comparison reports, code coverage trending reports.

JCov product

What documentation is available for developers?

JCov tutorial (in jcov repository) shows common methods of getting code coverage with JCov.

Why JCov is the desirable choice for OpenJDK developers?

JCov is interesting because it is developed in lock-step with the development of the Java platform (JDK). JCov always maintains the version of Java which is currently under development.
Also, JCov is well integrated with Oracle's current test infrastructure, such as JavaTest and the JTReg test harnesses.
This is important because JCov is focused on the processing of large volumes of heterogeneous tests and results.

What are the unique advantages of JCov?

- Off-line (so-called static) instrumentation as well as on-the-fly (so-called dynamic) instrumentation. This is an important feature; it allows you to choose the most effective mode for your coverage measurement task.
- JCov provides the ability to instrument JDK classes including early (bootstrap classes) core java classes.
- The network grabber and merger allow users to collect data from different sources, such as multiple concurrent tests in different JVMs or results different test run aggregations.
- JCov allows filtering of inner invocations
- JCov allows Diff analysis
- JCov allows Coverage of individual tests
- JCov allows Javap annotation of coverage results

What is the procedure to receive code coverage data for any product?

The general procedure to receive code coverage information for any product contains 3 general steps:

1. instrumentation of product code (that means adding invocations of code coverage tool methods in all product methods)
2. running tests using a previously instrumented product
3. save and display of the collected code coverage results

with JCov you can perform these steps separately (which is called static instrumentation), and simultaneously (which is dynamic instrumentation mode).

How do I get coverage of JRE or similar product?

The procedure to collect code-coverage data for a JRE (or similar product) is generally the same as any other product (same as the question above).

The following instructions will demonstrate how obtain coverage of a JRE build provided by a test suite. The kind of test suite doesn't matter (code example to get code coverage data for regression tests run by JTReg).

Step 1: Instrumentation

Prepare your own instrumented copy of a JRE. jreInstr command will do all the necessary work.

```
# cp -r  jdk-build-bNN  $JDK_INSTR
# java -jar jcov.jar jreinstr -rt jcov_network_saver.jar $JDK_INSTR/jre
```

Step 2: Starting the network grabber in a separate console (grabber is TCP socket server)

The network grabber will collect and merge coverage data sent by tests. It could be run on a remote machine.

```
# java -jar jcov.jar grabber -t template.xml -o result.xml &
```

Step 3: Running tests

Run the tests in the manner just specify your instrumented jre as the build to test.

If you ran the grabber on a remote host or used alternative port you need to set JCOV_HOST and JCOV_PORT env variables.

Run openJDK regression tests by JTReg:

```
# java -jar jtreg.jar -jdk $JDK_INSTR $JDK_WS/jdk/test
```

Step 4: Stopping the grabber

When test execution is completed give the following command:

```
# java -jar jcov.jar grabberManager -stop
```

The grabber will be stopped, the result XML coverage file produced.

Step 5: Generating coverage report

To see the report in a human readable form execute the repgen command:

```
# java -jar jcov.jar repgen -o report-bNN/ [-src jdk/src/] result.xml
```

How do I use JCov with jtreg?

Please visit jtreg support for jcov page

Code example to get code coverage data for regression tests run by JTReg:

```
# java -jar jtreg.jar -jdk:$JDK -w:$JDK_WS -r:rep -avm -jcov/classes:$JDK/jre/lib/rt.jar -jcov/source:$JDK/src.zip -jcov/include:java.math.* /test/*
```

What is the difference between static and dynamic modes?

Static and *Dynamic* are the terms used in the JCov realm to distinguish the moment of *when* classes are being instrumented.

In *static* mode, the application classes are instrumented in a separate process in advance, before test execution. The (JCov) *instr* command is used for that purpose.

In *dynamic* mode, the classes are instrumented while class-loading. (*-javaagent=jcov.jar*) No extra step is required to get coverage.

The mechanism of instrumentation is the same. The output format is the same as well.

	static	**dynamic**
result file size	as large as the template	much smaller
time	minor overhead on test execution might require time saving results	might takes significant time
changes in test suite	require adding jcov classes on classpath	require specifying extra vm option to -javaagent

In dynamic mode, the report doesn't include unused classes. To see real coverage, a merge with template is required. (use tmplGen command to generate the template)

The network grabber may help save time on result dumping.

Recommendations: how to choose the appropriate mode:

- dynamic - good for small and simple cases (eg. small apps, limited API, things that don't require a large set of Java classes to be loaded)
- static - more universal, easy to setup for complicated cases (eg. loading large API sets, things that require a large set of Java classes to be loaded)

Code example to get code coverage data for regression tests executed by JTReg in dynamic mode:

Step 1: Generate template

```
# java -jar jcov.jar tmplgen $JDK/jre
```

Step 2: Starting the network grabber

```
# java -jar jcov.jar grabber -t template.xml -o result.xml &
```

Step 3: Running tests

```
# java -jar jtreg.jar -jdk $JDK -vmoptions:"-javaagent:jcov.jar=grabber" -avm $JDK/jdk/test/*
```

Step 4: Stopping the grabber

```
# java -jar jcov.jar grabberManager -stop
```

Step 5: Generating coverage report

```
# java -jar jcov.jar repgen -o report -src $JDK/src.zip result.xml
```

Why can't I merge XML reports?

An attempt to merge two XML reports might be rejected by the merger command. In 99% of the cases, this happens because JCov finds incompatible differences in data. Possible situations when such an error could occur:

- coverage data obtained for two different versions of the applications (two jdk builds). E.g. in version 1 class A has two methods: a(), b(), in version 2 the same class has three methods: a(), b(), c(), where b() has been changed since the previous version
- one XML report could miss line table information (string like <lt>10=6;22=9</lt>). The same classes by complied in a different manner.
- one XML report was generated with some filter turned on (like skip abstract or synthetic methods), when another one was generated without filter

Merge error is a signal that you're trying to merge inconsistent data. If you're sure that the data you try to merge could be (and must be) merged you can try the following:

- specify "-loose 3" option, this will allow to ignore as much as possible inconsistencies
- merge with a template: java -jar jcov.jar -t template.xml r1.xml r2.xml

Is JCov making my Application/Test hang?

It's very unlikely that code inserted by jcov could make an application hang. The two most probable reasons if it does happen:

1. Test execution was terminated in the moment when jcov was saving data to file. Each time jcov creates a new file it creates a lock file to guarantee an exclusive access to the file. Unexpected termination might cause the situation when lock file hasn't removed. As the result, the next test execution will hang trying to allocate locked resource. Discovering files with names like resuslt.xml.fdsa-34vy-3434-FA3d-thtz after terminated execution should be a signal that not all lock files have been removed. Solution: remove all lock files (named like result.xml.lock), all created result file and files with long extensions. After that rerun tests
2. Tests are not hanged up, but takes extra time. Possible solutions:
    - Use grabber instead saving to file
    - Use include/exclude patterns to narrow the classes to collect coverage for

Is JCov producing malformed XML?

The most possible reason for malformed XML is that JCov processes have been terminated while saving data to a file. The solution is to use the JCov grabber - saving data to a file requires much more time than sending code coverage data to a JCov server.

How do I get code coverage information for individual tests in a test suite?

You can use the JCov extension jtobserver.jar (or write your own implementation of the javatest interface *Harness.Observer)* to receive total test coverage - as well as individual test coverage. Running tests with jtobserver will make JCov send data from the grabber after finishing each test. Sending testname for this covered data, the grabber will receive these coverage data from each test and merge them into one result.xml file - saving information about each tests coverage.

Here's a code example to get code coverage data for regression tests (from BigInteger package) run by JTReg with additional information for each test:

Step 1: Instrumentation

```
# cp -r  jdk-build-bNN  $JDK_INSTR
# java -jar jcov.jar jreinstr -rt jcov_network_saver.jar -i java.math.* $JDK_INSTR/jre
```

Step 2: Starting the network grabber

```
# java -jar jcov.jar grabber -t template.xml -o result.xml -outtestlist testlist &
```

Step 3: Running tests

```
# java -jar jtreg.jar -jdk $JDK_INSTR -vmoptions:"-Djcov.extension=jt" -o:jcov.JTObserver -od:jtobserver.jar -avm -conc:1 $JDK_INSTR/jdk/test/java
/math/BigInteger/*
```

Step 4: Stopping the grabber

```
# java -jar jcov.jar grabberManager -stop
```

Step 5: Generating coverage report

```
# java -jar jcov.jar repgen -o report -i java.math.* -tests testlist -testsinfo -src $JDK_INSTR/src.zip result.xml
```

What part of the covered data was tested purposely, and what was invoked as part of testing something else?

The feature of filtering inner invocations is based on the idea that it is important to see what methods are invoked explicitly from tests. This means that these methods are 'test goals', and the results of such methods are checked. This feature is implemented by creating additional classfile instrumentation: jcov will modify classfiles with additional invocations of the jcov method 'setExpected' for each method invocation found in the instrumented method.

```
Test code:
Timer t = new java.util.Timer();
But why AtomicInteger is covered?!
```

There are two different approaches to configure this additional instrumentation: you can specify all packages from where an invocation should be counted (caller_include caller_exclude options), or you can specify that you want to ignore inner invocations by setting '*innerinvations*' off.

With the first approach, you need to instrument not only the product but also tests - because you are interested in invocations from tests, but it will give you flexibility in configuring the invocation source.
With the second approach, no additional action is required, but you will get all invocations which could contain not only invocations from your tests - but also possibly from a library which is used in your product that has not been instrumented.

```
Instrumenting tests with -caller_include and
-caller_exclude options
Instrumenting product – ignore any calls inside JRE
>java -jar jcov.jar jreinstr -innerinvocation off …
```

Do we need a new test - or - how well does a particular change to the source code get covered by existing tests?

The common practice for developers before integrating their changes into a repository is to test the changes. Normally it's done by running existing tests. If the tests pass, it's assumed that change is safe to be integrated. But the problem is 'how to determine if the tests touch the changed code or not'. To answer the question: "how well is the change tested?" the following technique could be applied:

- run existing test with JCov to obtain result.xml file with coverage information
- get a patch file for the change
- use a JCov DiffCoverage utility to see which lines of changed code were not touched by tests

You can see the result of using DiffCoverage utility - some lines of code are marked as covered and we can see that tests have not tried to throw exceptions: blocked URL

Which execution branches are not tested?

The Javap feature is an additional report mode which allows you to use disassembled code instead of 'sources' as input to the report generation. By specifying the "javap" flag path to the RepGen command, you will receive a standard jcov html report with coverage information overlaid on javap disassembly for each class.
The main benefits of applying this approach:

- when sources are not available
- when there are two or more blocks in a line
- when no simple mapping between source code and generated code.

Also javap report could identify areas where new test cases are desired.

blocked URL