

Writing JPF Tests

[TracNav](#)

- [JPFWiki](#) - Welcome Page

[Introduction...](#)

[Installing JPF...](#)

[User Guide...](#)

[Developer Guide](#)

- [Design](#)
- [Choice Generator](#)
- [Partial Order Reduction](#)
- [Attributes](#)
- [Listener](#)

[MJL...](#)

- [Bytecode Factory](#)
- [Logging](#)
- [Report](#)
- [Embedded](#)
- [JPF tests](#)
- [JPF project layout](#)
- [Create a JPF project](#)
- [Coding Conventions](#)
- [Hosting update site](#)

[Projects...](#)

- [Summer Projects](#)
- [External Projects](#)
- [Change\(B\)log](#)

[About...](#)

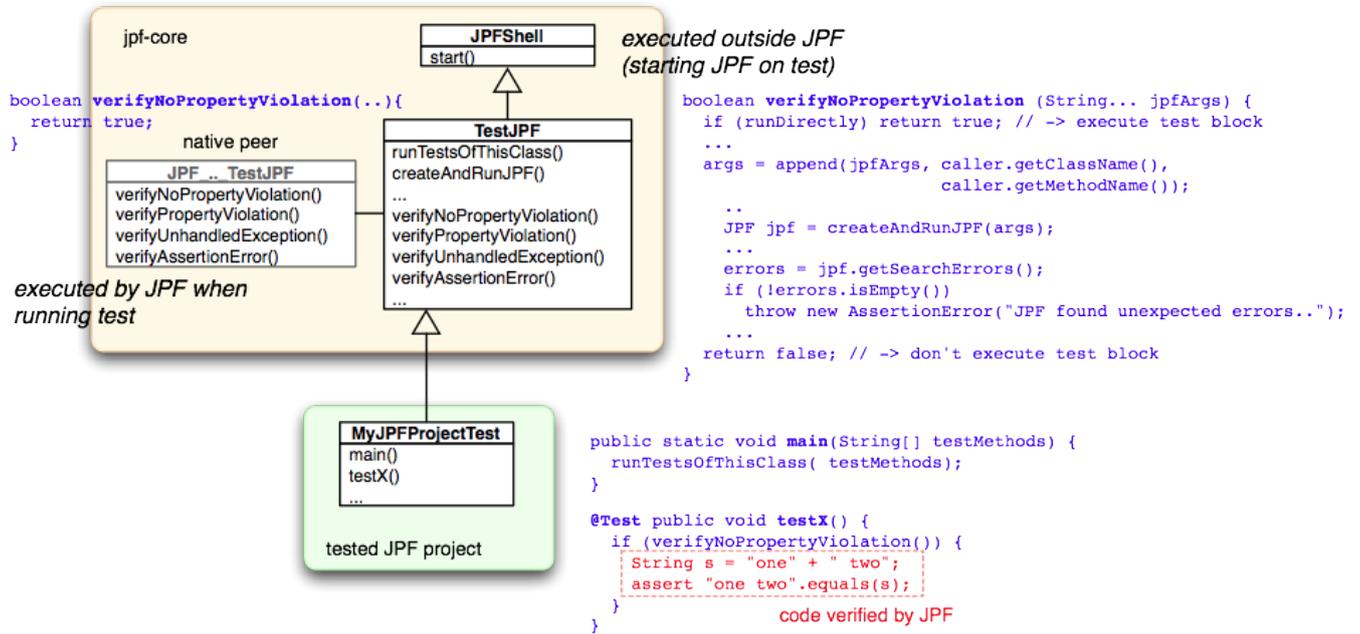
- [Events](#)
- [Presentations](#)
- [Papers](#)
- [FAQ](#)
- [History?](#)
- [Support](#)
- [People?](#)
- [Playground](#)
- [Table of Context](#)

As a complex runtime system for (almost) arbitrary Java programs, it goes without saying that JPF needs a lot of regression tests. You can find these under the `src/tests` directories in (hopefully) all JPF projects. All tests follow the same scheme, which is motivated by the need to run tests in a number of different ways:

1. as part of the Ant-based build system, i.e. from `build.xml`
2. as explicitly invoked JUnit tests
3. by directly running JPF on the test application (i.e. without JUnit, either as a `JPF shell` or via `RunTest.jar`)
4. by running the test application on a normal JVM

The rationale for this is to support various levels of inspection and debugging.

Each test conceptually consists of a test driver (e.g. executed under JUnit) which starts JPF from within its @Test annotated methods, and a class that is executed by JPF in order to check the verification goals. For convenience reasons, jpf-core provides infrastructure that enables you to implement both parts in the same class. This can be confusing at first - **the test class is used to start JPF on itself**.



The `main()` method of `TestJPF` derived classes always look the same and can be safely copied between tests:

```
public static void main(String[] testMethods){
    runTestsOfThisClass(testMethods);
}
```

This method serves two purposes. First, it is used to start the test outside JUnit, either on all @Test annotated instance methods, or just on the ones which names are provided as arguments. Second, it serves as the entry point for JPF when it executes the class. In this case, `TestJPF` takes care of invoking JPF on the test class and providing the name of the test method this was executed from.

Other than that, test classes just consist of (almost) normal @Test annotated JUnit test methods, which all share the same structure

```
import org.junit.Test;

@Test public void testX () {
    if (verifyNoPropertyViolation(JPF_ARGS){
        .. code to verify by JPF
    }
}
```

The trick is the call to `verifyNoPropertyViolation()`, or any of the other `verifyXX()` methods of `TestJPF`. If executed by the host VM, i.e. from JUnit, it starts JPF on the same class and the containing method, and returns `false`. This means the corresponding `if` block is **not** executed by the host VM.

When JPF is invoked, the argument to the `main()` method is set to the method name from which JPF got invoked, which causes `runTestsOfThisMethod()` to execute exactly this method again, but this time under JPF. Instead of re-executing the same `TestJPF.verifyX()` method again (and becoming infinitely recursive), we use a native peer `JPF_gov_nasa_jpf_util_test_TestJPF` which intercepts the `verifyX()` call and simply returns `true`, i.e. this time *only* the `if` block gets executed.

The rest of the host VM executed `TestJPF.verifyX()` checks the results of the JPF run, and accordingly throws an `AssertionError` in case it does not correspond to the expected result. The most common goals are

- `verifyNoPropertyViolation` - JPF is not supposed to find an error

- `verifyPropertyViolation` - JPF is supposed to find the specified property violation
- `verifyUnhandledException` - JPF is supposed to detect an unhandled exception of the specified type
- `verifyAssertionError` - same for `AssertionErrors`
- `verifyDeadlock` - JPF is supposed to find a deadlock

Each of these methods actually delegate running JPF to a corresponding method whose name does not start with 'verify..'. These workhorse methods expect explicit specification of the JPF arguments (including SUT main class name and method names), but they return JPF objects, and therefore can be used for more sophisticated JPF inspection (e.g. to find out about the number of states).

`TestJPF` also provides some convenience methods that can be used within test methods to find out which environment the code is executed from:

- `isJPFRun()` - returns true if the code is executed under JPF
- `isJUnitRun()` - returns true if the code is executed under JUnit by the host VM
- `isRunTestRun()` - returns true if the code is executed by `RunTest.jar`

Here is an example of a typical test method that uses some of these features:

```

@Test public void testIntFieldPerturbation() {

    if (!isJPFRun()){ // run this outside of JPF
        Verify.resetCounter(0);
    }

    if (verifyNoPropertyViolation("+listener=.listener.Perturbator",
                                "+perturb.fields=data",
                                "+perturb.data.class=.perturb.IntOverUnder",...
                                "+perturb.data.delta=1")){

        // run this under JPF
        System.out.println("instance field perturbation test");

        int d = data; // this should be perturbed
        System.out.println("d = " + d);

        Verify.incrementCounter(0);

    } else { // run this outside of JPF
        assert Verify.getCounter(0) == 3;
    }
}

```

Running JPF tests from command line

To run JPF tests from the command line, use the `RunTest.jar` either from `jpf-core/build`, or the one that is distributed with your project containing the tests (`tools/RunTest.jar` for JPF projects). This is an executable jar that expects the test class and (optional) method test names as arguments. If no method names are provided, all `@Test` annotated methods are executed. Most projects have a convenience script `bin/test` to execute `RunTest.jar`.

```

> bin/test gov.nasa.jpf.test.mc.data.PerturbatorTest testIntFieldPerturbation

..... testing testIntFieldPerturbation
  running jpf with args: +listener=.listener.Perturbator +perturb.fields=data +perturb.data.class=.perturb.IntOverUnder +pe
JavaPathfinder v5.x - (C) RIACS/NASA Ames Research Center

===== system under test
application: gov/nasa/jpf/test/mc/data/PerturbatorTest.java
arguments:   testIntFieldPerturbation

===== search started: 9/10/10 7:03 PM
instance field perturbation test

```

```

d = 43
d = 42
...
===== search finished: 9/10/10 7:03 PM
..... testIntFieldPerturbation: Ok

..... execution of testsuite: gov.nasa.jpf.test.mc.data.PerturbatorTest SUCCEEDED
.... [1] testIntFieldPerturbation: Ok
..... tests: 1, failures: 0, errors: 0

```

Running JPF tests under JUnit

This is the preferred way to execute JPF regression tests, which is usually done from an Ant `build.xml` script containing a standard target such as

```

...
<target name="test" depends="build" description="run core regression tests" if="have_tests">
  ...
  <junit printsummary="on" showoutput="off" haltonfailure="yes"
        fork="yes" forkmode="perTest" maxmemory="1024m" outputtoformatters="true">
    <classpath>
      <path refid="lib.path"/>
      <pathelement location="build/tests"/>
      ...
    </classpath>
    <batchtest todir="build/tests">
      <fileset dir="build/tests">
        <exclude name="**/JPF_*.class"/>
        <include name="**/*Test.class"/>
      </fileset>
    </batchtest>
  </junit>
</target>

```

Most JPF projects have `build.xml` files you can use as examples.

Please note this means that you should not have any inner classes, interfaces, annotation types etc. that end with `*Test` since JUnit would interpret these as test cases and most likely complain about missing constructors and `main()` methods.

Debugging tests

Typically, JPF tests are only executed from within an IDE if they fail and need to be debugged.

Under NetBeans, this can be done by selecting the test class, and then executing the *Debug File* command from the context menu. This will pop up a dialog that lets you enter a specific test method to debug. This method requires a properly set up `ide-file-target.xml`, which comes with most JPF projects.

Under Eclipse, you can select the test class and then execute *Debug As.. -> Java Application*.