

Wikiprint Book

Title: Understanding JPF Output

Subject: Java Path Finder - user/output

Version: 3

Date: 02/17/13 23:58:27

Table of Contents

Understanding JPF Output	3
TracNav	3
Introduction...	3
Installing JPF...	3
User Guide	3
Developer Guide...	3
Projects...	3
About...	3
Application Output	3
Logging	4
Reports	4

Understanding JPF Output

[TracNav](#)

- [JPFWiki](#) - Welcome Page

[Introduction...](#)

[Installing JPF...](#)

[User Guide](#)

- [Application Types](#)
- [JPF Components](#)
- [Configuring JPF](#)
- [Running JPF](#)
- [JPF Output](#)
- [The JPF API](#)

[Developer Guide...](#)

[Projects...](#)

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

[About...](#)

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

There are three different ways a JPF run can produce output, each of them with a different purpose, but all controlled by the `[[user:config|general JPF configuration mechanism]]`:

1. application output - what is the application doing?
2. JPF logging - what is JPF doing?
3. JPF reporting system - what is the result of the JPF run?

Application Output

This is the most simple form of output, which usually just consists of `System.out.println(...)` calls embedded in the application code. There is only one caveat - since this is executed by JPF as part of the application, the same print statement might be executed several times:

```
public class MyApplication ..{
    ...
    boolean cond = Verify.getBoolean();
    System.out.println("and the cond is: " + cond);
    ...
}
```

will produce

```
...
and the cond is: true
...
and the cond is: false
...
```

The second execution of the print statement is of course preceded by a backtrack operation of JPF (the *Verify.getBoolean()* statement has two choices {true,false}), but the backtracking might not be visible, e.g. when running without the *ExecTracker* or *ChoiceTracker* listeners.

Since it can be sometimes confusing to see the same output twice without knowing if there is an iteration in the application, or JPF did backtrack between executions, there are two configuration options to control the output behavior:

vm.tree_output={true|false} - means output is shown on the console each time a print statement is executed. This corresponds to the above example, and is the default behavior.

vm.path_output={true|false} - will not immediately print the output on the console, but store in the path for subsequent processing once JPF terminates (if the *output* topic is specified - see below). This should produce the same output as running the test application on a normal JVM.

Logging

This is a more interesting form of JPF output, primarily intended to show what JPF does internally. For this purpose, it has to support various levels of details, ranging from severe errors to fine grained logging of JPF operations.

JPF's logging mechanism does not reinvent the wheel, it piggybacks on the standard java.util.logging infrastructure. While this means it would be possible to use customized LogHandlers and Formatters (e.g. to log in XML format), there are specialized JPF incarnations of these classes, mainly to enable logging configuration via the standard JPF configuration mechanism rather than system properties.

Using the JPF Logging involves two aspects: (1) controlling log output destination, and (2) setting log levels. Both are done with JPF property files.

To set the default log level, use the *log.level* property (the supported levels being *severe*, *warning*, *info*, *fine*, *finer*, *finest*)

If you want to log to a different console that possibly even runs on a remote machine, use the *gov.nasa.jpff.tools.LogConsole* on the machine that should display the log messages:

```
$ java gov.nasa.jpff.tools.LogConsole <port>
```

Then start JPF on the test machine, specifying where the log output should go:

```
$ jpf +log.output=<host>:<port> ... MyTestApp
```

The default host is "localhost", default port is 20000. If these are suitable settings, you can start the *LogConsole* without parameters, and just specify *+log.output=socket* when running JPF.

If you develop your own JPF classes, please also check the [\[\[devel:logging|JPF logging API\]\]](#) page.

Reports

The JPF reporting system is used to show the outcome of a JPF run, to report property violations, print traces, show statistics and much more. This is in a way the most important part of the JPF user interface, and might involve various different output formats (text, XML, API calls) and targets (console, IDE). Depending on application and project, users might also need control over what items are displayed in which order. It is also obvious this needs to be an extensible mechanism, to adapt to new tools and properties. The JPF report system provides all this, again controlled by JPF's general configuration mechanism.

The basic concept is that reporting depends on a predefined set of output phases, each of them with a configured, ordered list of topics. The output phases supported by the current system are:

- **start** - processed when JPF starts
- **transition** - processed after each transition
- **property_violation** - processed when JPF finds a property violation
- **finished** - processed when JPF terminates

There is no standard *transition* topic yet (but it could be implemented in PublisherExtensions). The standard *property_violation* topics include:

- **error** - shows the type and details of the property violation found
- **trace** - shows the program trace leading to this property violation
- **snapshot** - lists each threads status at the time of the violation
- **output** - shows the program output for the trace (see above)
- **statistics** - shows property statistics information

Last not least, the finished list of topics that usually summarizes the JPF run:

- **result** - reports if property violations were found, and shows a short list of them
- **statistics** - shows overall statistics information

The system consists of three major components: (1) the Reporter, (2) any number of format specific Publisher objects, and (3) any number of tool-, property- and Publisher-specific PublisherExtension objects. Here is the blueprint:

Again, there is separate [report system API](#) documentation if you are interested in JPF development.