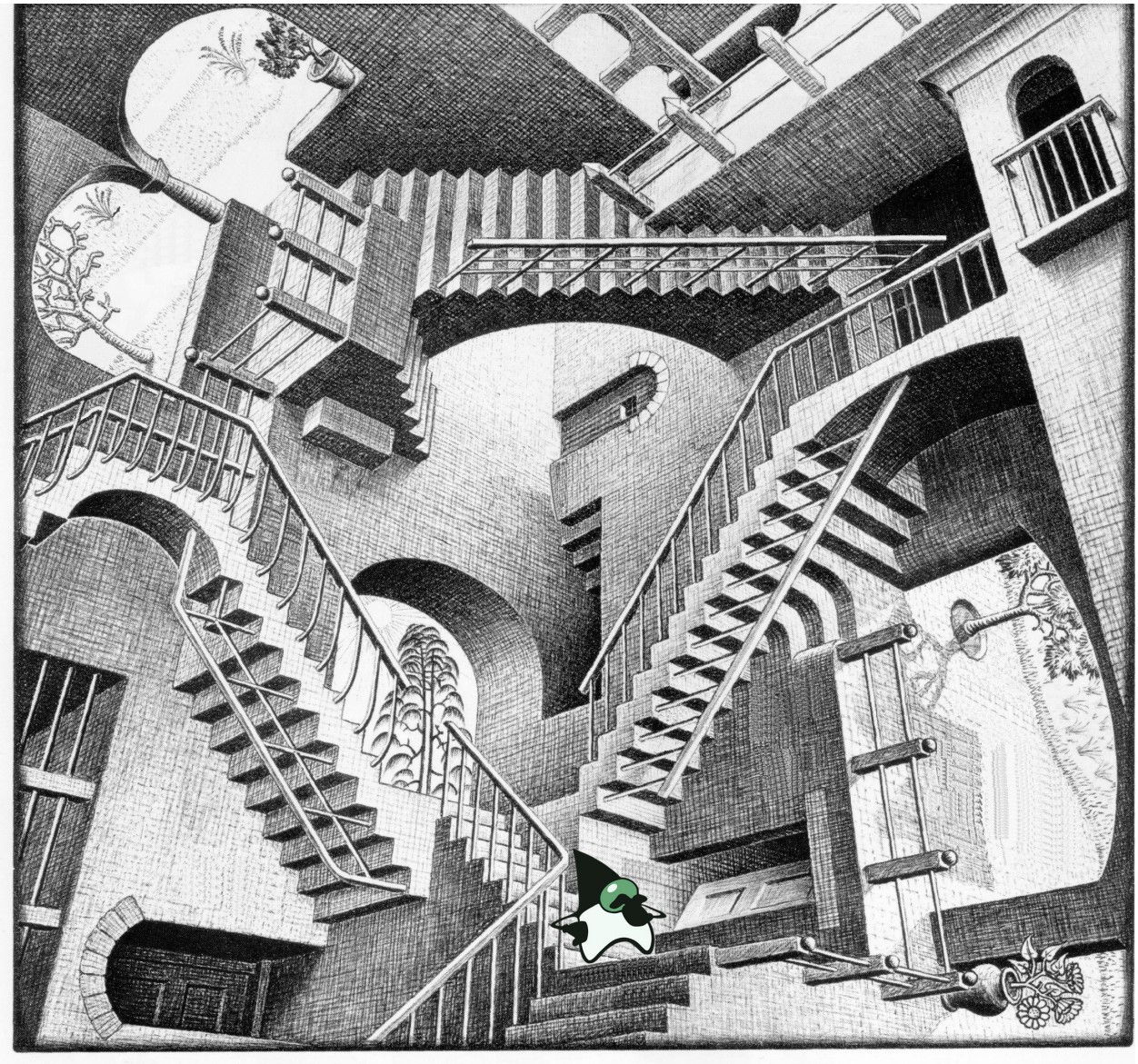
# Java PathFinder: a tool to detect bugs in Java code



Franck van Breugel

# Abstract

It is well known that software contains bugs. Since Java is among the most popular programming languages, it is essential to have tools that can detect bugs in Java code. Although testing is the most used technique to detect bugs, it has its limitations, especially for nondeterministic code. Concurrency and randomization are the two main sources of nondeterminism. To find bugs in nondeterministic code, testing needs to be complemented with other techniques such as model checking. Java PathFinder (JPF) is the most popular model checker for Java code. In this book, we describe how to install, configure, run and extend JPF.

# Contents

To Lisa

# Preface

According to a 2002 study commissioned by the US Department of Commerce's National Institute of Standards and Technology, "estimates of the economic costs of faulty software in the US range in the tens of billions of dollars per year and have been estimated to represent approximately just under one percent of the nation's gross domestic product." In 2013, a study [BJC+13] concluded that "wages-only estimated cost of debugging is US $312 billion per year." Since software development has not changed drastically in the last decade, but the footprint of software in our society has increased considerably, it seems reasonable to assume that this number has increased as well and ranges in trillions of dollars per year on a world wide scale. This was confirmed by a recent study in 2018 by Krasner [Kra18] which observed that "the cost of poor quality software in the US in 2018 is approximately $2.84 trillion." Hence, *tools to detect bugs* in software can impact the software industry and even the world economy. The topic of this book is such a tool.

The TIOBE programming community index[1], the transparent language popularity index[2], the popularity of programming language index[3], the RedMonk programming language rankings[4], and Trendy Skills[5], all rank *Java* among the most popular programming languages. Popularity of the language and impact of a tool to detect bugs of software written in that language go hand in hand. Therefore, we focus on a popular language in this book, namely Java.

*Testing* is the most commonly used method to detect bugs. However, for *nondeterministic* code testing may be less effective. Code is called nondeterministic if it gives rise to different executions even when all input to the code is fixed. Randomization and concurrency both give rise to nondeterminism. Since concurrency is generally considered more intricate than randomization, our examples will predominantly focus on the latter. Chapter **??** will concentrate on concurrency. To illustrate the limitations of testing when it comes to nondeterministic code, let us consider the following Java application.

```
1 import java.util.Random;
2
3 public class Example {
4   public static void main(String[] args) {
5     Random random = new Random();
6     System.out.print(random.nextInt(10));
7   }
8 }
```

The above application may result in ten different executions, since it prints a randomly chosen integer in the interval $[0, 9]$. Now, let us replace line 6 with

```
    System.out.print(1 / random.nextInt(9));
```

In 80% of the cases, the application prints zero, in 10% it prints one, and in the remaining 10% it crashes because of an uncaught exception due to a division by zero. Of course, it may take more than ten executions before we encounter the exception. In case we choose an integer in the interval $[0, 999999]$ it may take many executions before encountering

[1] www.tiobe.com
[2] lang-index.sourceforge.net
[3] pypl.github.io/PYPL.html
[4] redmonk.com/sogrady/2018/08/10/language-rankings-6-18/
[5] trendyskills.com

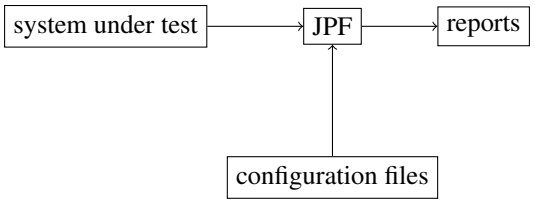



Figure 1: Overview of JPF.

the exception. If we execute the application one million times, there is still a 37% chance that we do not encounter the exception.[6]

In the presence of nondeterminism, testing does not guarantee that all different executions are checked. Furthermore, if a test detects a bug in nondeterministic code, it may be difficult to reproduce. Therefore, in that case methods which complement testing are needed. *Model checking* is such an alternative. It aims to check all potential executions of nondeterministic code in a systematic way.

We will not discuss model checking in much detail. Instead, we refer the interested reader to textbooks such as [BK08], [CGK+18] and [BBF+01]. In this book, we introduce the reader to a model checker, a tool that implements model checking. In particular, we focus on a model checker for Java.

Although there are several model checkers for Java, including Bandera [CDH+00] and Bogor [RDH03] to name a few, *Java PathFinder* (JPF) is the most popular one. Its popularity is reflected by several statistics. For example, the conference paper [VHBP00] and its extended journal version [VHB+03] have been cited more than 1800 times according to Google scholar, making it the most cited work on a Java model checker. In this book, we focus on JPF. Although JPF can do much more than detect bugs, we concentrate on that functionality.

## Overview of JPF

In Figure 1 we provide a high level overview of JPF. It takes as input a system under test and configuration files and produces reports as output. The *system under test* is the application, a Java class with a `main` method, we want to check for bugs. JPF not only checks that `main` method but also all other code that is used by that `main` method. JPF can only check a closed system, that is, a system for which all input is provided, be it obtained from the keyboard, the mouse, a file, a URL, et cetera.

JPF can be configured in two different ways: by command line arguments or in configuration files. We will concentrate on the second option. There are three different types of configuration file. We will discuss them in Chapter 3.

The reports that JPF produces can take different forms. For example, a report can be written to the console or to a file, and it can be text or XML. In the configuration files one can specify what type of reports should be produced by JPF. We will discuss this in more detail in Chapter **??**.

## Overview of the Book

This book has been written for both students and developers who are interested in tools that can help them with detecting bugs in their Java code. In Chapter 1 we discuss how to install JPF. How to run JPF is the topic of Chapter 2. In Chapter we focus on the configuration of JPF.

[6] The probability of choosing zero is $\frac{1}{1000000}$. The probability of not choosing zero is $1 - \frac{1}{1000000} = \frac{99999}{1000000}$. The probability of not choosing zero one million times in a row is $(\frac{99999}{1000000})^{1000000} \approx 0.37$.

# Acknowledgements

First of all, I want to thank the team at NASA Ames Research Center for developing and maintaining JPF. Without them, there would be no book.

Secondly, I want thank those who have worked with me on JPF: Yuvaraj Anbarasan, Cyrille Artho, Matteo Ceccarello, Vivek Chaudhari, Syyeda Zainab Fatmi, Sergey Kulikov, Saad Naji, Nastaran Shafiei, Qiyi Tang, Mateusz Ujma, Willem Visser, Steven Xu, Zeyu Xu, and Xin Zhang. Furthermore, I would like to thank Rami Abou-Nassar and Xiang Chen for detailed feedback on parts of the book.

Finally, I want to thank the students who took the course in which I used drafts of this book: Ayman Abualsunun, Davood Anbarnam, Alexander Aolaritei, Kevin Arindaeng, Jonathan Bahri, Yash Chauhan, Eric Dao, Dev Dutta, Rabia Ejaz, Adham El Shafie, Andrew Ferreira, Asma Hassan, Yahya Ismail, Abasifreke James, Shagun Kazan, Jason Kuffour, Skyler Layne, Derek Li, Vladimir Martintsov, Daniel McVicar, Drew Noel, Ante Pimentel, Amgad Rady, Varsha Ragavendran, Siraj Rauff, Behshad Sebthosseini, Nisha Sharma, Dan Sheng, Glib Sitiugin, Anton Sitkovets, Artem Solovey, Danilo Torres Fleites, Adrian Winkler, and Mina Zaki.

# Chapter 1

# Installing JPF

As we have already discussed in the preface, JPF is a tool to detect bugs in Java code. Since the reader is interested in JPF, we feel that it is safe to assume that the reader is familiar with Java and has installed the Java development kit (JDK). The JDK should be version 8. Use the latest update of version 8, that is, 1.8.0_281, as JPF relies on some methods that are not present in earlier updates such as 1.8.0_251. JPF can be installed in several different ways on a variety of operating systems. A road map for Section 1.1–1.8 can be found in Figure 1.1.

Since changes are made to JPF on a regular basis, it is best to obtain its souces from JPF's GitHub repository. GitHub a web-based hosting service for version control. Information about GitHub can be found at github.com. We describe three different ways to install (and update) the sources of JPF's GitHub repository: au naturel, within Eclipse, and within NetBeans, in Section 1.1 (and 1.2), 1.3 (and 1.4), and 1.6 (and 1.7), respectively. For those using Eclipse or NetBeans, the latter two options are more convenient. Also, there are JPF plugins for Eclipse or NetBeans. How to install those is discussed in Section 1.5 and 1.8, respectively. How to use these plugins to run JPF is discussed in Chapter 2.

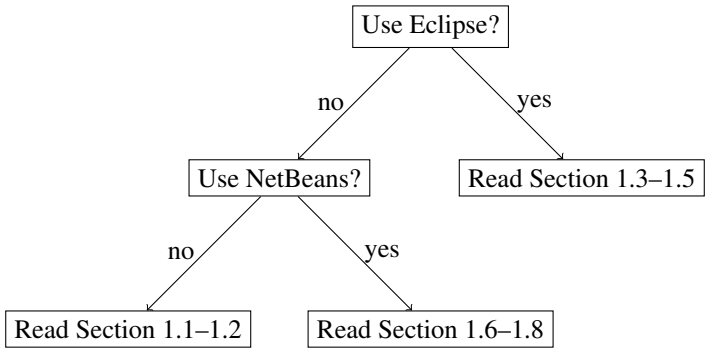



Figure 1.1: Road map for Section 1.1–1.8.

As we already mentioned in the preface, JPF is easily extensible. Therefore, it should come as no surprise that there are numerous extensions of JPF. In Section 1.9 we will discuss how to install such an extension.

## 1.1 Installing Sources with Git

How to install Git is beyond the scope of this book. We refer the reader to git-scm.com. We assume that the path to the `git` command is already part of the system environment variable `PATH` (see Section 1.1.3). To install the JPF sources with Git, follow the seven steps below.

1. Create a directory named `jpf`.

2. To get the JPF sources with Git, open a shell (Linux and OS X) or command prompt (Windows), go to the `jpf` directory and type

   ```
   git clone https://github.com/javapathfinder/jpf-core.git
   ```

   This results in output similar to the following.

   ```
   Cloning into 'jpf-core'...
   remote: Enumerating objects: 2784, done.
   remote: Total 2784 (delta 0), reused 0 (delta 0), pack-reused 2784
   Receiving objects: 100% (2784/2784), 1.86 MiB | 12.19 MiB/s, done.
   Resolving deltas: 100% (1324/1324), done.
   ```

3. Set the user environment variable `JAVA_HOME` as described in Section 1.1.1.

4. Run the Gradle script as described in Section 1.1.2.

5. Set the user environment variable `JPF_HOME` to the path of `jpf-core`. For example, if the `jpf` directory, created in step 1, has path `/cs/home/franck/jpf`, then the path of `jpf-core` is `/cs/home/franck/jpf/jpf-core`. Similarly, if the `jpf` directory has path `C:\Users\ franck\jpf`, then the path of `jpf-core` is `C:\Users\franck\jpf\jpf-core`.

6. Add the path of the `jpf` command to the system environment variable `PATH` as described in Section 1.1.3.

7. Create the `site.properties` file as described in Section 1.1.4.

Once the above steps have been successfully completed, the reader can move on to Chapter 2 and run JPF.

### 1.1.1 Setting User Environment Variable `JAVA_HOME`

**Linux**

1. Locate the directory of the JDK. Unless the install path for the JDK was changed during installation, it will be a subdirectory of `/usr/java`. Inside that directory will be one or more subdirectories whose name starts with `jdk`, such as `jdk1.8.0_181`. Choose the latest version. For example, if the directory contains both `jdk1.6.0_37` and `jdk1.8.0_181`, then the JDK install path is `/usr/java/jdk1.8.0_181`.

2. Set the user environment variable named `JAVA_HOME` to the directory of the JDK by using the `set` or `setenv` command in a startup script. For more details, do a web search for how to set an environment variable in Linux.

**Windows**

1. Locate the directory of the JDK. Unless the install path for the JDK was changed during installation, it will be a subdirectory of `C:\Program Files\Java`. Inside that directory will be one or more subdirectories whose name starts with `jdk`, such as `jdk1.8.0_181`. Choose the latest version. For example, if the directory contains both `jdk1.6.0_37` and `jdk1.8.0_181`, then the JDK install path is `C:\Program Files\Java\jdk1.8.0_181`.

2. Set the user environment variable named `JAVA_HOME` to the directory of the JDK. For more details, do a web search for how to set an environment variable in Windows.

**OS X**

1. Locate the directory of the JDK. Unless the install path for the JDK was changed during installation, it will be a subdirectory of `/Library/Java/JavaVirtualMachines`. Inside that directory will be one or more subdirectories whose name starts with `jdk`, such as `jdk1.8.0_181`. Choose the latest version. For example, if the directory contains both `jdk1.6.0_37` and `jdk1.8.0_181`, then the JDK install path is `/Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home`.
2. Set the user environment variable named `JAVA_HOME` to the directory of the JDK by using the `set` or `setenv` command in a startup script. For more details, do a web search for how to set an environment variable in OS X.

### 1.1.2 Running Gradle Script

**Linux and OS X**

In a shell, go to the subdirectory `jpf-core` of the created directory `jpf`. The directory `jpf-core` contains the file `gradlew`. To run this Gradle script, type `./gradlew`. This results in a lot of output, including something similar to the following.

```
> Task :compileJava
...

> Task :compileClassesJava
...

> Task :compilePeersJava
...

> Task :compileTestJava
...

BUILD SUCCESSFUL in 41s
15 actionable tasks: 15 executed
```

**Windows**

In a command prompt, go to the subdirectory `jpf-core` of the created directory `jpf`. The directory `jpf-core` contains the file `gradlew.bat`. To run this Gradle script, type `gradlew`. This results in a lot of output, including something similar to the following.

```
> Task :compileJava
...

> Task :compileClassesJava
...

> Task :compilePeersJava
...

> Task :compileTestJava
...

BUILD SUCCESSFUL in 53s
15 actionable tasks: 15 executed
```

### 1.1.3 Adding to the System Environment Variable `PATH`

The system environment variable `PATH` consists of a list of directories in which programs are located. Below, we discuss how to add the directory containing the `jpf` command can be added to `PATH`.

#### Linux

Add to the system environment variable named `PATH` the directory of the `jpf` command by changing the `set` or `setenv` command for `PATH` in a startup script. If the `jpf` directory has path `/cs/home/franck/jpf`, then add `/cs/home/franck/jpf/jpf-core/bin` to the system environment variable `PATH`. For more details, do a web search for how to change an environment variable in Linux.

#### Windows

In Windows, environment variables are not case sensitive. Hence, the system environment variable `PATH` can also be named, for example, `Path` or `path`. If the `jpf` directory has path `C:\Users\franck\jpf`, then add `C:\Users\franck\jpf\jpf-core\bin` to the system environment variable `PATH`. For more details, do a web search for how to change an environment variable in Windows.

#### OS X

If the `jpf` directory has path `/Users/franck/jpf`, then add `/Users/franck/jpf/ jpf-core/bin` to the system environment variable `PATH`. For more details, do a web search for how to change an environment variable in OS X.

### 1.1.4 Creating the `site.properties` File

1. Find the value of the standard Java system property *user.home* by running the following Java application.

```
public class PrintUserHome {
  public static void main(String[] args) {
    System.out.println("user.home = " + System.getProperty("user.home"));
  }
}
```

2. Create a directory named `.jpf` within the directory *user.home*[1].

3. Create in the directory *user.home*`/.jpf` a file named `site.properties`[2]. Assuming, for example, that `jpf-core` is a subdirectory of *user.home*`/jpf`, the file `site.properties` has the following content.

```
# JPF site configuration
jpf-core=${user.home}/jpf/jpf-core
extensions=${jpf-core}
```

Next, we provide a few examples.

[1] To create a directory named `.jpf` in Windows Explorer, use `.jpf.` as its name. The dot at the end is necessary, and will be removed by Windows Explorer.

[2] To create a file named `site.properties` in Window Explorer, configure Windows Explorer so that file extensions are visible, create a text file named `site.txt` with the above content, and rename the file to `site.properties`. For more details, do a web search for how to change a file extension in Windows.

**Linux**

Assume that the `jpf` directory has path `/cs/home/franck/jpf` and *user.home* is `/cs/home/ franck`. Then `site.properties` is located in the directory `/cs/home/franck/.jpf` and its content is

```
# JPF site configuration
jpf-core=${user.home}/jpf/jpf-core
extensions=${jpf-core}
```

If the `jpf` directory has path `/cs/packages/jpf` and *user.home* is `/cs/home/franck`, then `site.properties` is located in the directory `/cs/home/franck/.jpf` and its content is

```
# JPF site configuration
jpf-core=/cs/packages/jpf/jpf-core
extensions=${jpf-core}
```

**Windows**

Assume that the `jpf` directory has path `C:\Users\franck\jpf` and *user.home* is `C:\Users\ franck`. Then `site.properties` is located in the directory `C:\Users\franck\.jpf` and its content is

```
# JPF site configuration
jpf-core=${user.home}/jpf/jpf-core
extensions=${jpf-core}
```

Note that we use `/` instead of `\` in the path. If the `jpf` directory has path `C:\Program Files\jpf` and *user.home* is `C:\Users\franck`, then `site.properties` is located in the directory `C:\Users\franck\.jpf` and its content is

```
# JPF site configuration
jpf-core=C:/Program Files/jpf/jpf-core
extensions=${jpf-core}
```

**OS X**

Assume that the `jpf` directory has path `/Users/franck/jpf` and *user.home* is `/Users/franck`. Then `site.properties` is located in the directory `/Users/franck/.jpf` and its content is

```
# JPF site configuration
jpf-core=${user.home}/jpf/jpf-core
extensions=${jpf-core}
```

If the `jpf` directory has path `/System/Library/jpf` and *user.home* is `/Users/franck`, then `site.properties` is located in the directory `/Users/franck/.jpf` and its content is

```
# JPF site configuration
jpf-core=/System/Library/jpf/jpf-core
extensions=${jpf-core}
```

## 1.2 Updating Sources with Git

Since the sources of JPF change regularly, one should update JPF regularly as well. This can be done as follows.

1. Open a shell (Linux and OS X) or command prompt (Windows), go to the `jpf-core` directory and type

   ```
   git pull
   ```

We distinguish two cases. If the above command results in output similar to the following, then the sources of JPF have not changed and, hence, we are done.

```
Already up to date.
```

Otherwise, the above command results in output similar to the following, which indicates that the source of JPF have changed and, therefore, we continue with the next step.

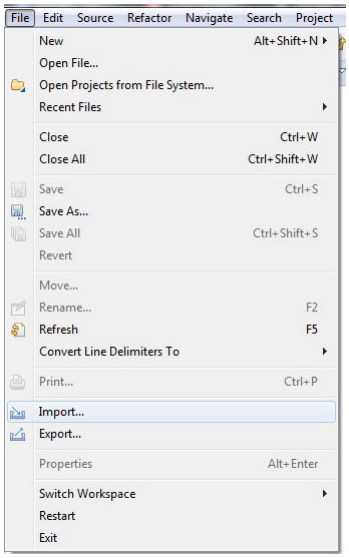
```
remote: Enumerating objects: 48, done.
remote: Counting objects: 100% (48/48), done.
remote: Compressing objects: 100% (19/19), done.
remote: Total 48 (delta 29), reused 42 (delta 23), pack-reused 0
Unpacking objects: 100% (48/48), done.
From https://github.com/javapathfinder/jpf-core
  28c066f..c42e4fc master -> origin/master
Updating 28c066f..c42e4fc
Fast-forward
 build.gradle                                       | 24 ++++++--
 .../gov/nasa/jpf/vm/JPF_java_lang_Runtime.java     |  8 +++
 .../gov/nasa/jpf/test/java/lang/RuntimeTest.java | 16 +++++
 5 files changed, 191 insertions(+), 7 deletions(-)
```

2. Run the Gradle script as described in Section 1.1.2.

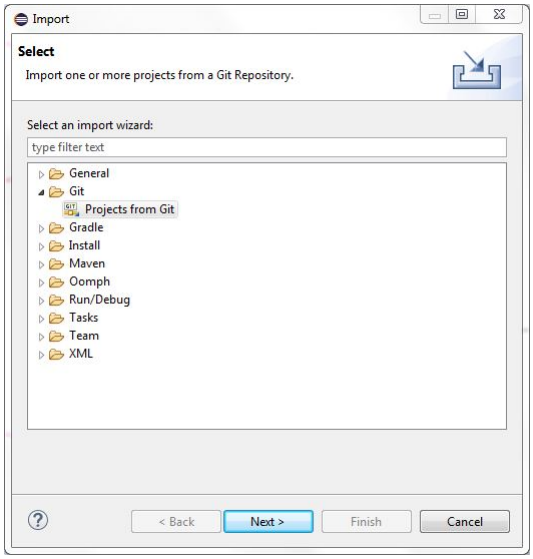
## 1.3 Installing Sources with Git within Eclipse

How to install Eclipse and Git is beyond the scope of this book. We refer the reader to eclipse.org and git-scm.com, respectively. We assume that Eclipse and Git have been installed. Eclipse should at least be version 4.6 and it should use Java version 8. We assume that the path to the `git` command is already part of the system environment variable `PATH` (see Section 1.1.3). To install the JPF sources with Git within Eclipse, follow the ten steps below.

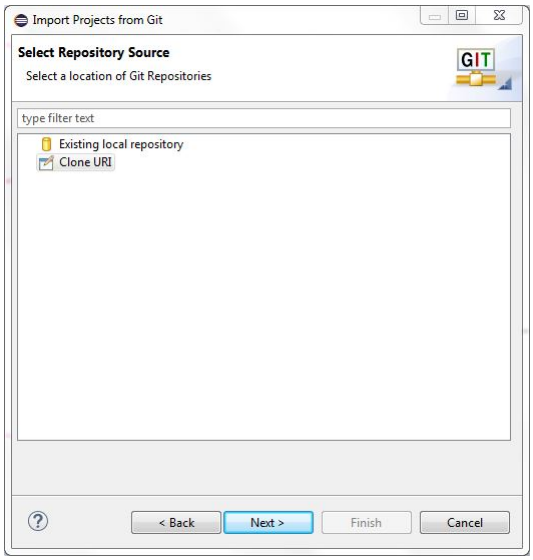
1. In Eclipse, select File from the menu and select Import...



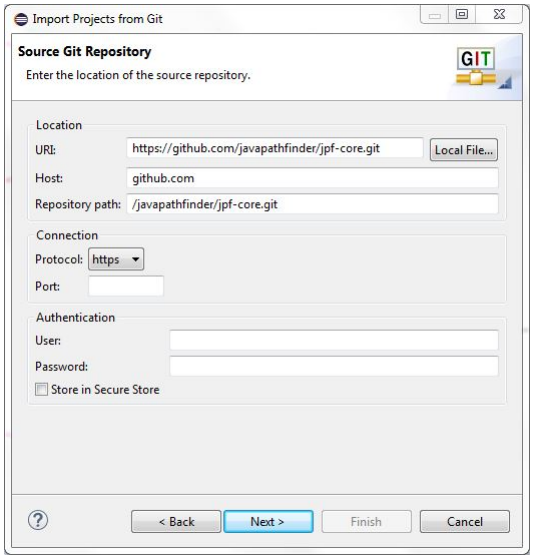

2. In the Select window, select Projects from Git, and press Next.



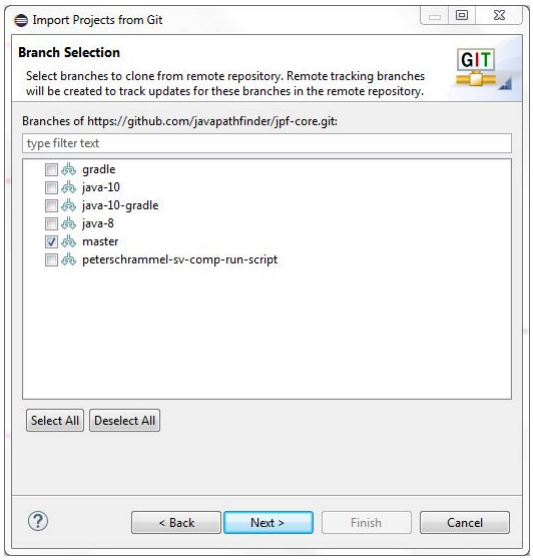

3. In the Select Repository Source window, select Clone URI, and press Next.



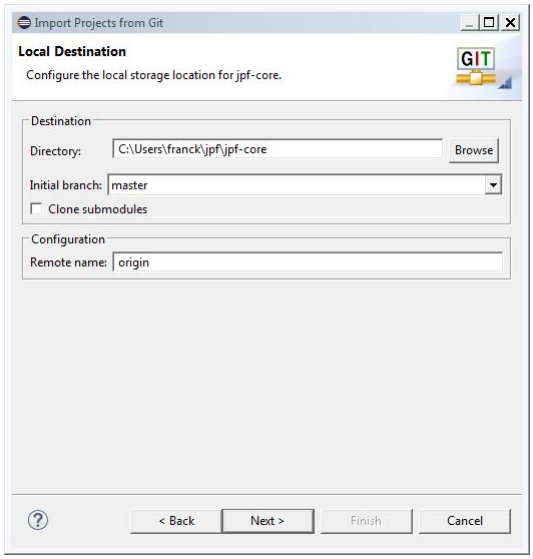

4. In the Source Git Repository window, enter `https://github.com/javapathfinder/jpf-core.git` in the URI field, and press Next. The fields Host and Repository path are populated automatically.



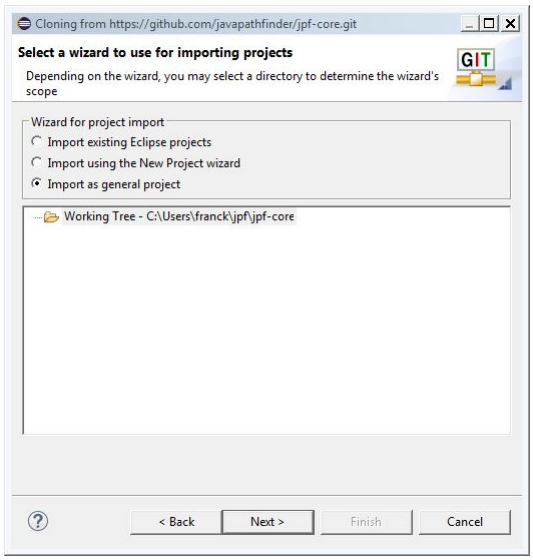

5. In the Branch Selection window, select master, and press Next.



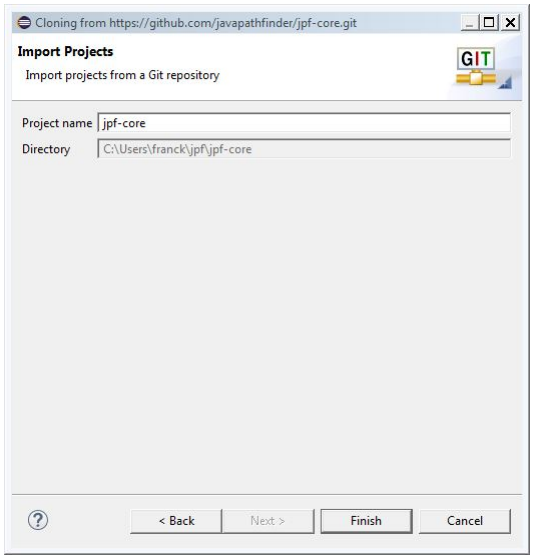

6. In the Location Destination window, modify the Directory to `C:\Users\franck\jpf\jpf-core`, that is, replace `git` with `jpf`, and press Next.




7. In the wizard selection window, select Import as general project, and press Next.




8. In the Imports window, click Finish.




9. Run the Gradle script as described in Section 1.1.2.

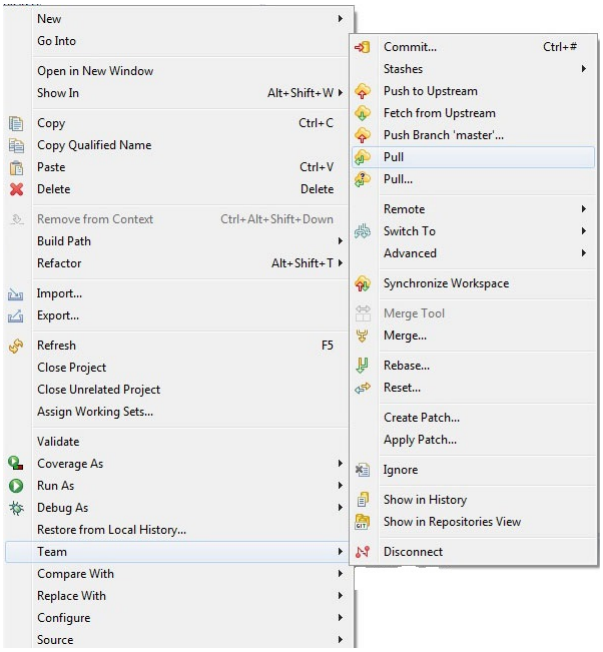
10. Create the `site.properties` file as described in Section 1.1.4. Assuming that the `jpf-core` directory has path `C:\Users\franck\jpf\jpf-core` and *user.home* is `C:\Users\franck`, the content of `site.properties` is

```
# JPF site configuration
jpf-core=${user.home}/jpf/jpf-core
extensions=${jpf-core}
```

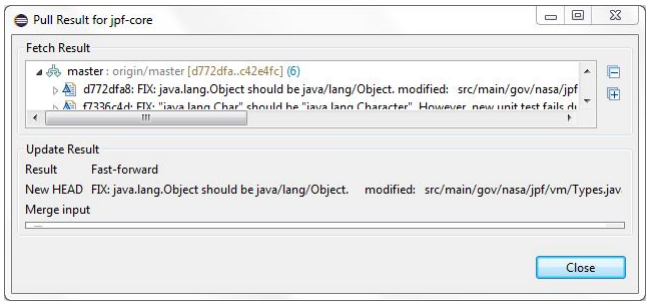
## 1.4 Updating Sources with Git within Eclipse

To update the sources with Git within Eclipse, follow the three steps below.

1. Right click on the jpf-core project in the Package Explorer and select Team and subsequently select Pull.




2. In the Pull Result window, click Close.




3. Run the Gradle script as described in Section 1.1.2.

## 1.5 Installing JPF Plugin for Eclipse

As we will discuss in Chapter 2, the JPF plugin can be used to run JPF within Eclipse. This plugin unfortunately does not work with the latest version of Eclipse.

### Linux

1. Locate the directory of Eclipse. Unless the install path for Eclipse was changed during installation, it will be `/usr/local`. This directory contains a directory named `dropins`. In this directory `dropins`, create a directory named `plugins`.

2. Download the file `eclipse-jpf.jar` from www.eecs.yorku.ca/~franck/jpf. Save the jar file in the created `plugins` directory.

### Windows

1. Locate the directory of Eclipse. Unless the install path for Eclipse was changed during installation, it will be `C:\Program Files\eclipse`. This directory contains a directory named `dropins`.

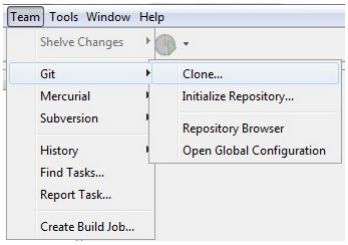
2. Download the file `eclipse-jpf.jar` from www.eecs.yorku.ca/∼franck/jpf. Save the jar file in the created `dropins` directory.

**OS X**

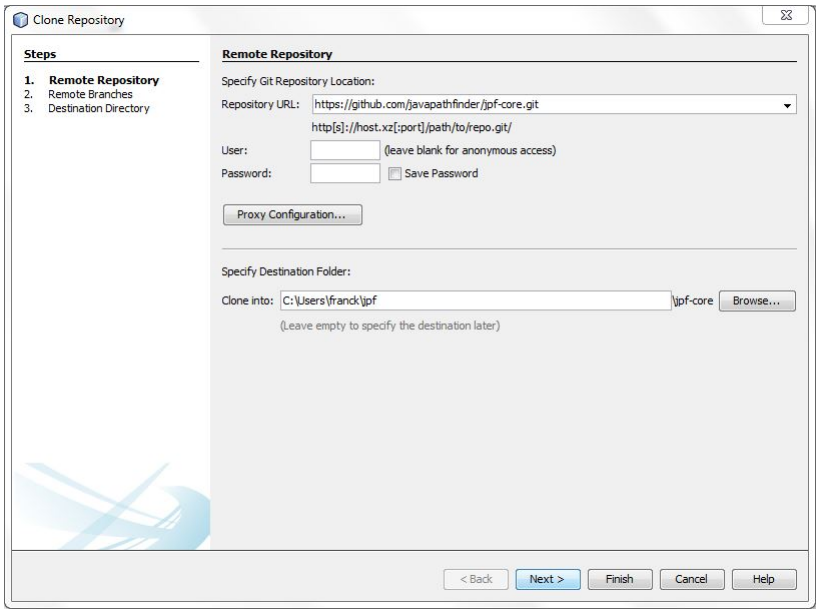
## 1.6 Installing Sources with Git within NetBeans

How to install NetBeans and Git is beyond the scope of this book. We refer the reader to netbeans.org and git-scm.com, respectively. We assume that NetBeans and Git have been installed. NetBeans should at least be version 8.0 and it should use Java version 8. To install the JPF sources with Git within NetBeans, follow the ten steps below.

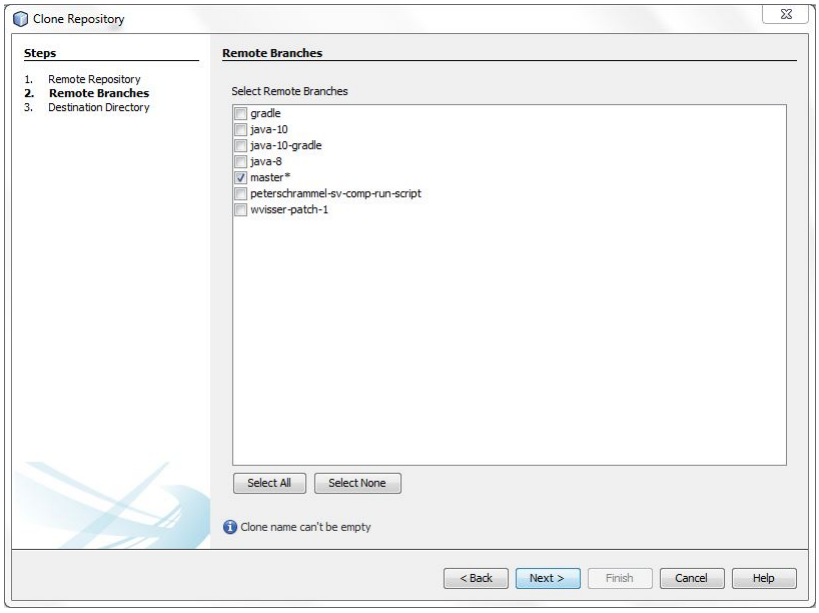
1. In NetBeans, select Team from the menu and select Clone...



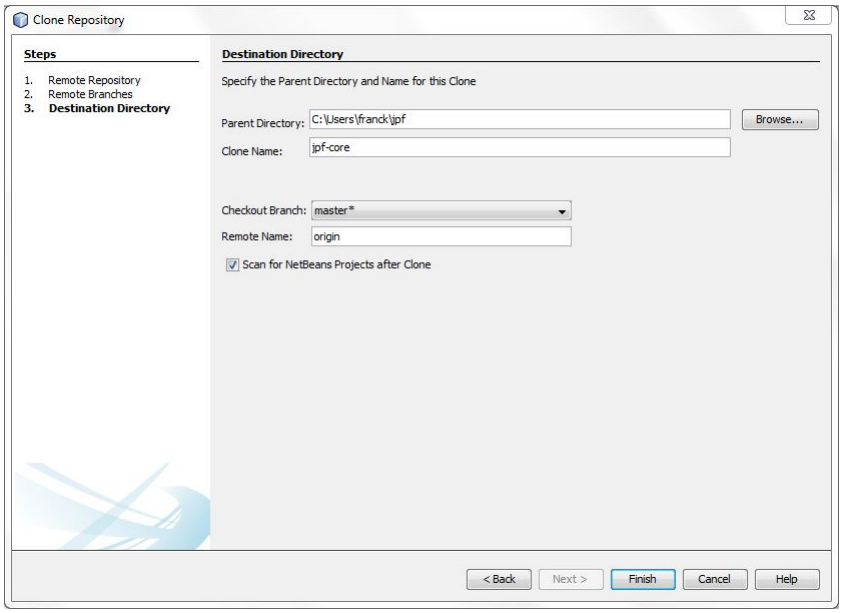

2. In the Clone Repository window, enter the Repository URL `https://github.com/javapathfinder/jpf-core.git`, set Clone into to `C:\Users\franck\jpf`, and press Next.



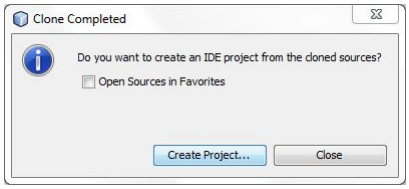

3. In the Clone Repository window, select the master branch, and press Next.




4. In the Clone Repository window, press Finish.

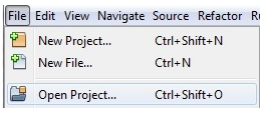



5. In the Clone Completed window, press Close.

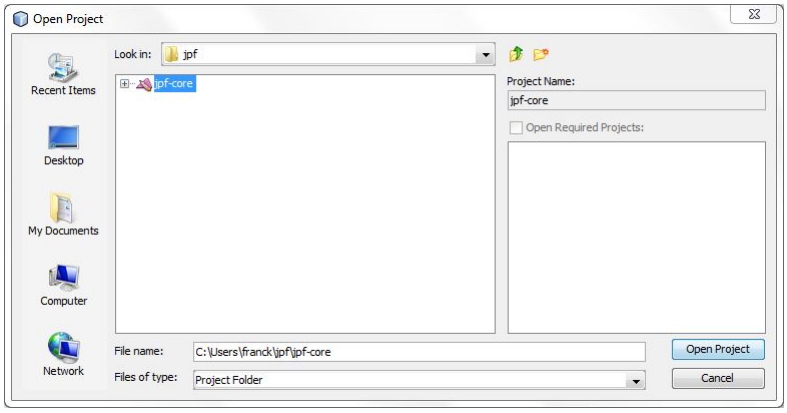



6. Download the file `netbeans.zip` from www.eecs.yorku.ca/~franck/jpf. Save the file in the directory `C:\Users\franck\jpf\jpf-core`. Extract all the files from `netbeans-jpf.zip`.

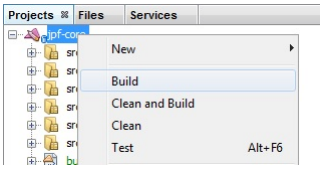
7. In NetBeans, select File from the menu and select Open Project...




8. In the Open Project window, select the `jpf-core` project in the `C:\\Users\franck\jpf` directory, and press Open Project.




9. Right click on the `jpf-core` project and select Build.



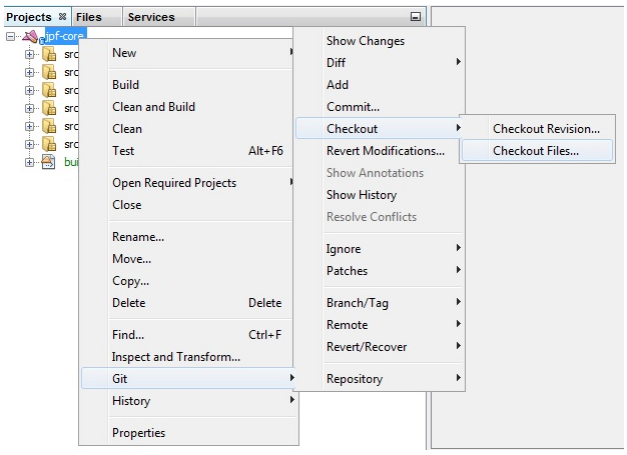

10. Create the `site.properties` file as described in Section 1.1.4. Assuming that the `jpf-core` project has path `C:\Users\franck\jpf` and *user.home* is `C:\Users\franck`, the content of `site.properties` is

```
# JPF site configuration
jpf-core=${user.home}/jpf/jpf-core
extensions=${jpf-core}
```

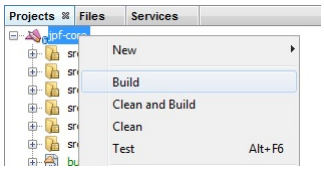
## 1.7 Updating Sources with Git within NetBeans

To update the sources with Git within NetBeans, follow the two steps below.

1. Right click on the `jpf-core` project, select Git, Checkout, and Checkout Files.




2. Right click on the `jpf-core` project and select Build.

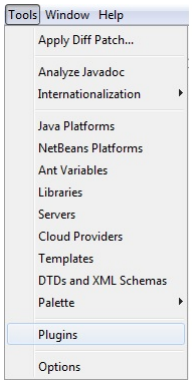



## 1.8 Installing JPF Plugin for NetBeans

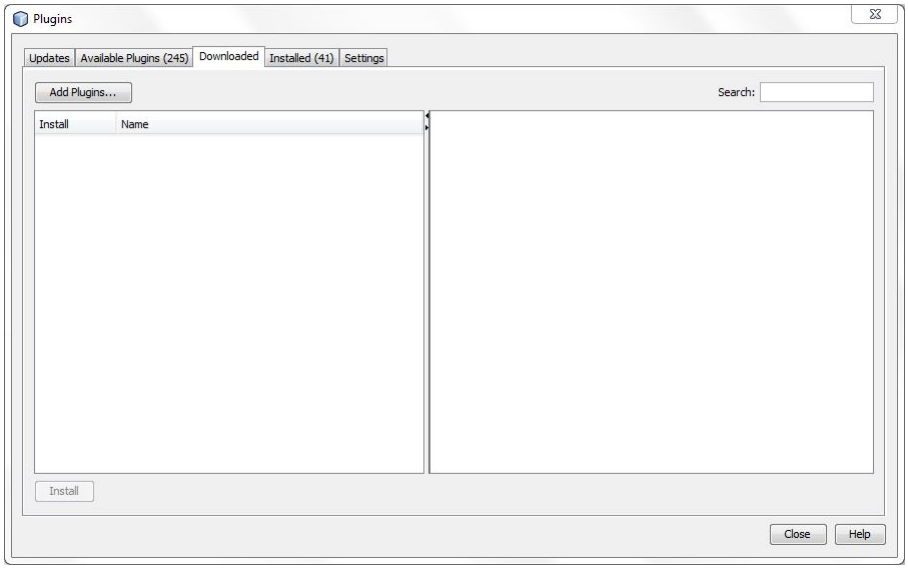
As we will discuss in Chapter 2, the JPF plugin can be used to run JPF within NetBeans. This plugin can be installed as follows.

1. Download the file github.com/javapathfinder/jpf-core/wiki/install/gov-nasa-jpf-netbeans-runjpf.nbm.

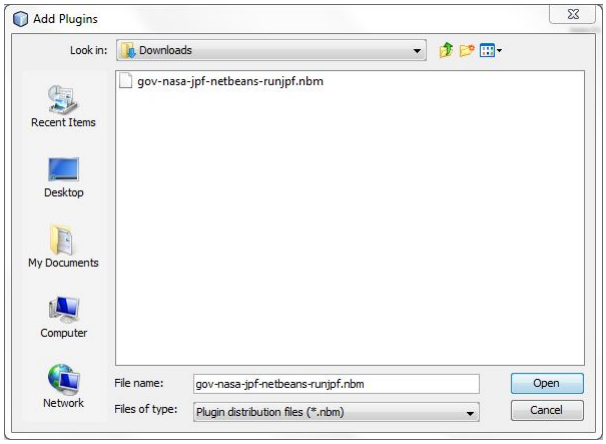
2. In NetBeans, select Tools from the menu and the select Plugins.



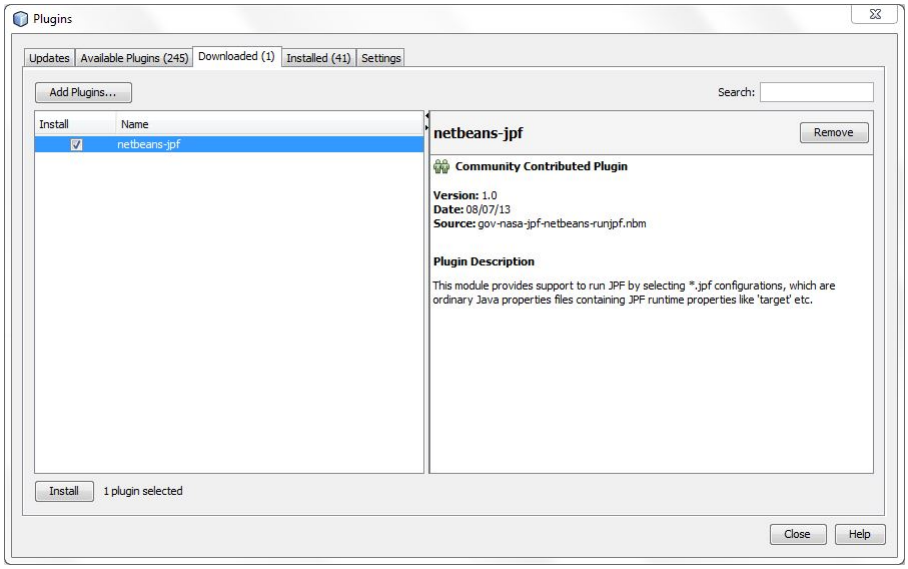

3. Select the Downloaded tab. Press on Add Plugins...



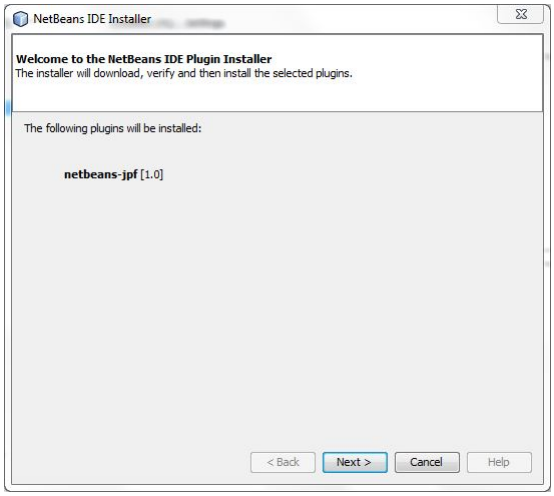

4. Select the `gov-nasa-jpf-netbeans-runjpf.nbm` file that was downloaded in step 1, and press Open.



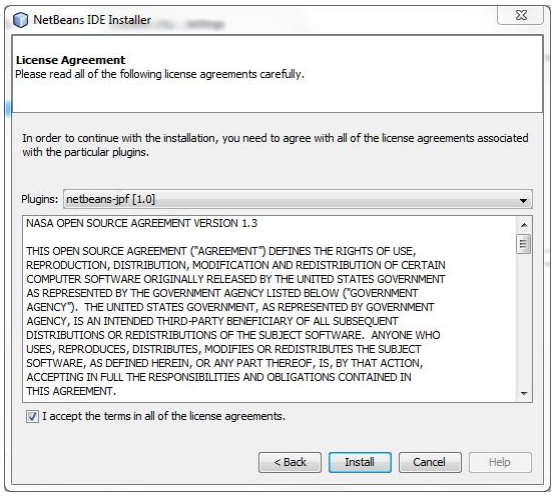

5. In the Plugins window, press Install.




6. In the NetBeans IDE Installer window, press Next.




7. In the NetBeans IDE Installer window, accept the license agreement, and press Install.




8. Restart NetBeans.

## 1.9 Installing an Extension of JPF

As running example, we consider the extension `jpf-shell`. This extension provides a graphical user interface for JPF. We assume that the reader has already successfully installed `jpf-core`.

### 1.9.1 Installing Sources with Mercurial

How to install Mercurial is beyond the scope of this book. We refer the reader to www.mercurial-scm.org. We assume that the path to the `hg` command is already part of the system environment variable `PATH` (see Section 1.1.3). To install the JPF sources with Mercurial, follow the three steps below.

1. To get the `jpf-shell` sources with Mercurial, open a shell (Linux and OS X) or command prompt (Windows), go to the `jpf` directory, assumed here to be `C:\Users\franck\jpf`, and type

   ```
   hg clone https://jpf.byu.edu/hg/jpf-shell
   ```

   This results in output similar to the following.

   ```
   destination directory: jpf-shell
   requesting all changes
   adding changesets
   adding manifests
   adding file changes
   added 194 changesets with 953 changes to 232 files (+1 heads)
   new changesets f3ac7d4188ec:c785dfae0b01
   updating to branch default
   131 files updated, 0 files merged, 0 files removed, 0 files unresolved
   ```

2. Run `ant` as described in Section 1.9.2.

3. Update the `site.properties` file as described in Section 1.9.3.

### 1.9.2 Running Ant

Ant is a Java library and command-line tool that can be used to compile the sources of JPF and its extensions, test them, generate jar files, et cetera. For more information about Ant, we refer the reader to ant.apache.org. We assume that the reader has already installed Ant (see ant.apache.org for installation instructions) and has added the `ant` command to the `PATH` environment variable (see Section 1.1.3).

For example, to install the JPF extension `jpf-shell`, open a shell (Linux and OS X) or command prompt (Windows) and go to the subdirectory `jpf-shell` of the directory `jpf`. The directory `jpf-shell` contains the file `build.xml`. To run `ant`, type `ant test`. This results in a lot of output, the beginning and end of which are similar to the following.

```
Buildfile: C:\Users\franck\jpf\jpf-shell\build.xml

-init:
[mkdir] Created dir: C:\Users\franck\jpf\jpf-shell\build

...

-min:
[jar] Building jar: C:\Users\franck\jpf\jpf-shell\build\jpf-shell-min.jar

build:

BUILD SUCCESSFUL
Total time: 13 seconds
```

### 1.9.3 Updating the `site.properties` File

Whenever we install an extension of JPF, we need to update the `site.properties` file.

#### Linux

Assume that the `jpf` directory has path `/cs/home/franck/jpf` and *user.home* is `/cs/home/ franck`. Then `site.properties` is located in the directory `/cs/home/franck/.jpf` and its content is

```
# JPF site configuration
jpf-core=${user.home}/jpf/jpf-core
jpf-shell=${user.home}/jpf/jpf-shell
extensions=${jpf-core}
```

If the `jpf` directory has path `/cs/packages/jpf` and *user.home* is `/cs/home/franck`, then `site.properties` is located in the directory `/cs/home/franck/.jpf` and its content is

```
# JPF site configuration
jpf-core=/cs/packages/jpf/jpf-core
jpf-shell=/cs/packages/jpf/jpf-shell
extensions=${jpf-core}
```

#### Windows

Assume that the `jpf` directory has path `C:\Users\franck\jpf` and *user.home* is `C:\Users\ franck`. Then `site.properties` is located in the directory `C:\Users\franck\.jpf` and its content is

```
# JPF site configuration
jpf-core=${user.home}/jpf/jpf-core
jpf-shell=${user.home}/jpf/jpf-shell
extensions=${jpf-core}
```

Note that we use / instead of \ in the path. If the `jpf` directory has path `C:\Program Files\jpf` and *user.home* is `C:\Users\franck`, then `site.properties` is located in the directory `C:\Users\franck\.jpf` and its content is

```
# JPF site configuration
jpf-core=C:/Program Files/jpf/jpf-core
jpf-shell=C:/Program Files/jpf/jpf-shell
extensions=${jpf-core}
```

#### OS X

Assume that the `jpf` directory has path `/Users/franck/jpf` and *user.home* is `/Users/franck`. Then `site.properties` is located in the directory `/Users/franck/.jpf` and its content is

```
# JPF site configuration
jpf-core=${user.home}/jpf/jpf-core
jpf-shell=${user.home}/jpf/jpf-shell
extensions=${jpf-core}
```

If the `jpf` directory has path `/System/Library/jpf` and *user.home* is `/Users/franck`, then `site.properties` is located in the directory `/Users/franck/.jpf` and its content is

```
# JPF site configuration
jpf-core=/System/Library/jpf/jpf-core
jpf-shell=/System/Library/jpf/jpf-shell
extensions=${jpf-core}
```

### 1.9.4 Updating Sources with Mercurial

To update the sources with Mercurial, follow the two steps below.

1. Open a shell (Linux and OS X) or command prompt (Windows), go to the `jpf-shell` directory and type

   ```
   hg pull -u
   ```

   We distinguish two cases. If the above command results in output similar to the following, then the sources of `jpf-shell` have not changed and, hence, we are done.

   ```
   pulling from https://jpf.byu.edu/hg/jpf-shell
   searching for changes
   no changes found
   ```

   Otherwise, the above command results in output similar to the following, which indicates that the source of `jpf-shell` have changed and, therefore, we continue with the next step.

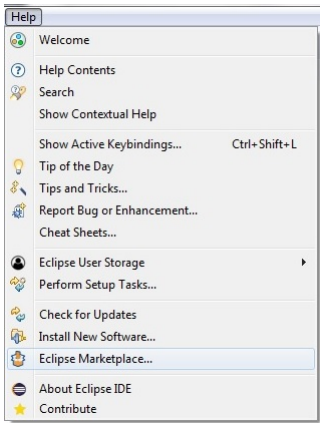
   ```
   pulling from https://jpf.byu.edu/hg/jpf-shell
   searching for changes
   adding changesets
   adding manifests
   adding file changes
   added 4 changesets with 23 changes to 7 files
   7 files updated, 0 files merged, 1 files removed, 0 files unresolved
   ```

2. Run `ant` as described in Section 1.9.2.

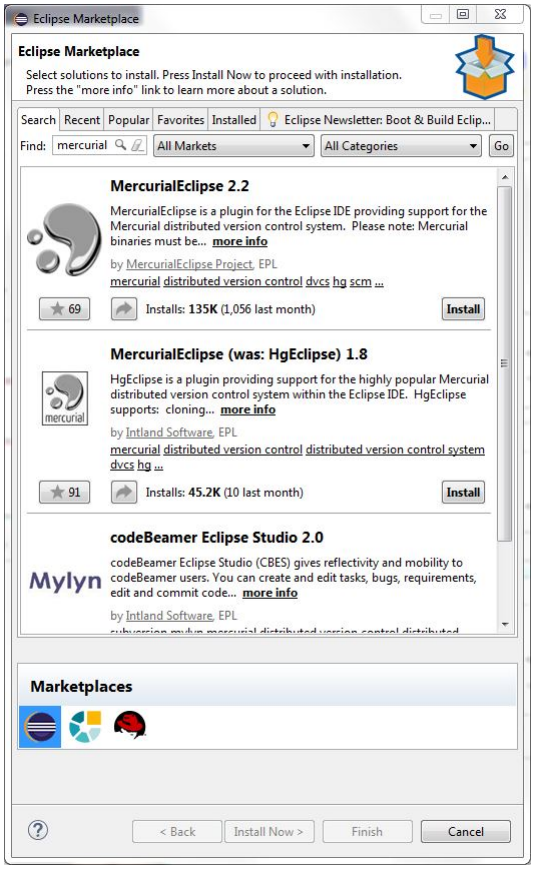
### 1.9.5 Installing Sources with Mercurial within Eclipse

How to install Mercurial is beyond the scope of this book. We refer the reader to www.mercurialscm.org. We assume that the path to the `hg` command is already part of the system environment variable `PATH` (see Section 1.1.3). To install the JPF extension `jpf-shell` within Eclipse with Mercurial, follow the 11 steps below.

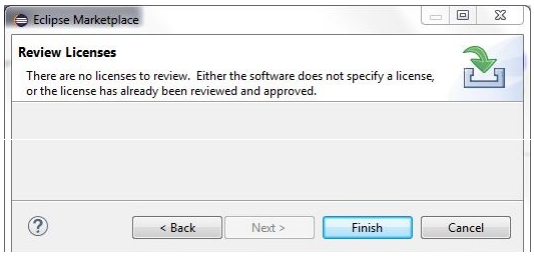
1. In Eclipse, select Help from then menu item and then select Eclipse Marketplace ...



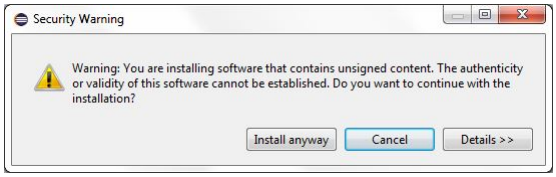

2. In the Eclipse Marketplace window, enter mercurial in the Find textbox, hit enter, and press Install.



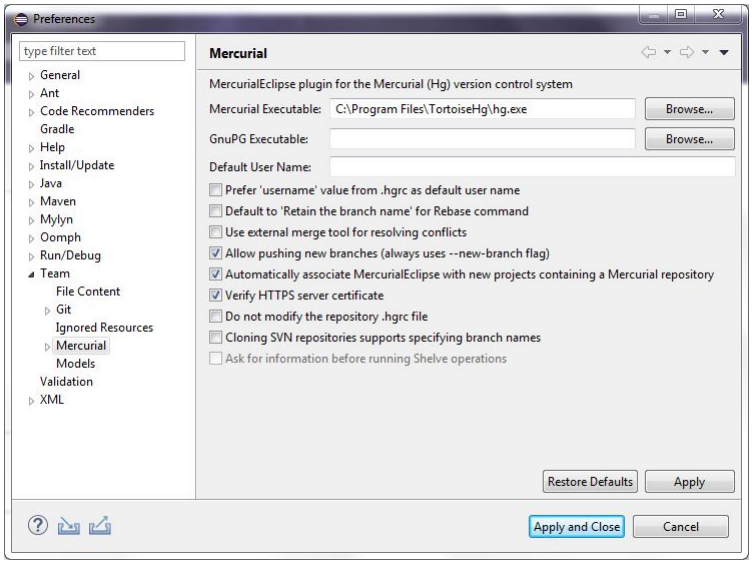

3. In the Review Licences window, press Finish.



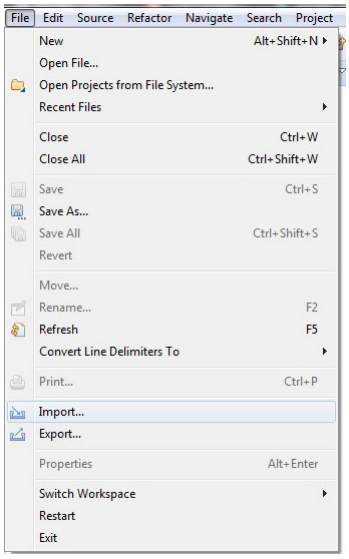

4. In the Security Warning window, press Install anyway.



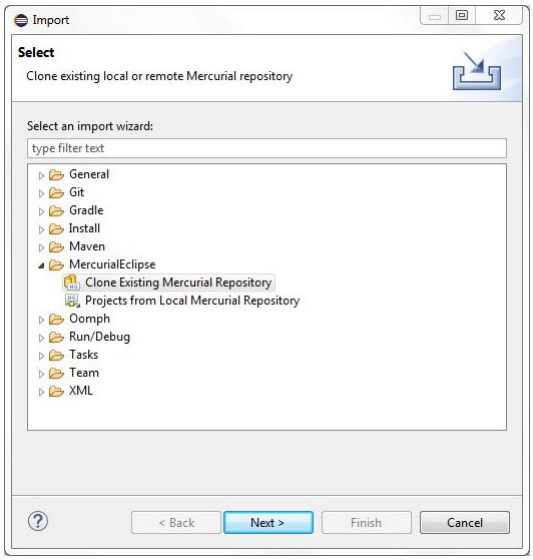

5. After Eclipse restarts, in the Preferences window, check that the path to the Mercurial Executable is correct and press Apply and Close.



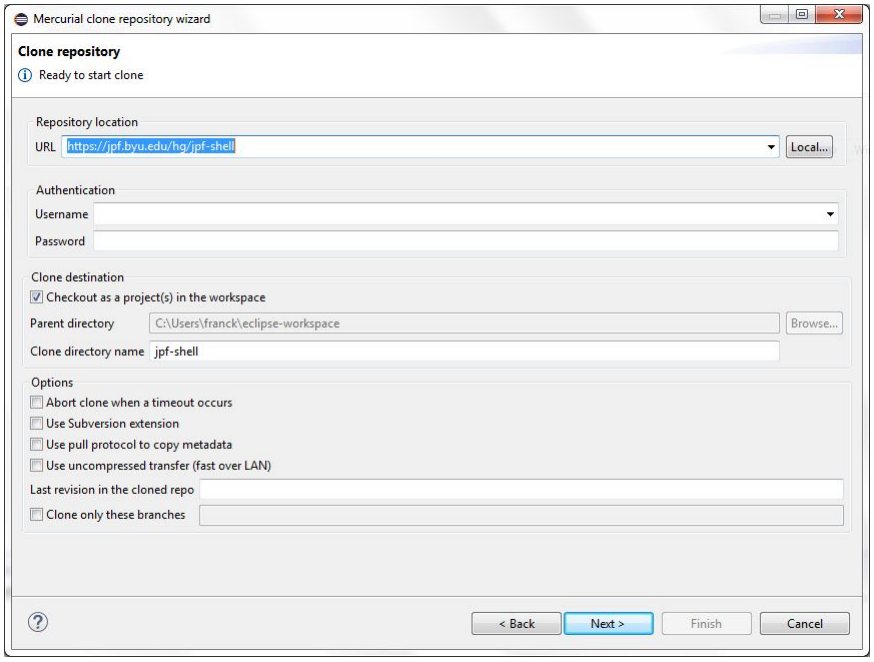

6. In Eclipse, select File from the menu item and then select Import.



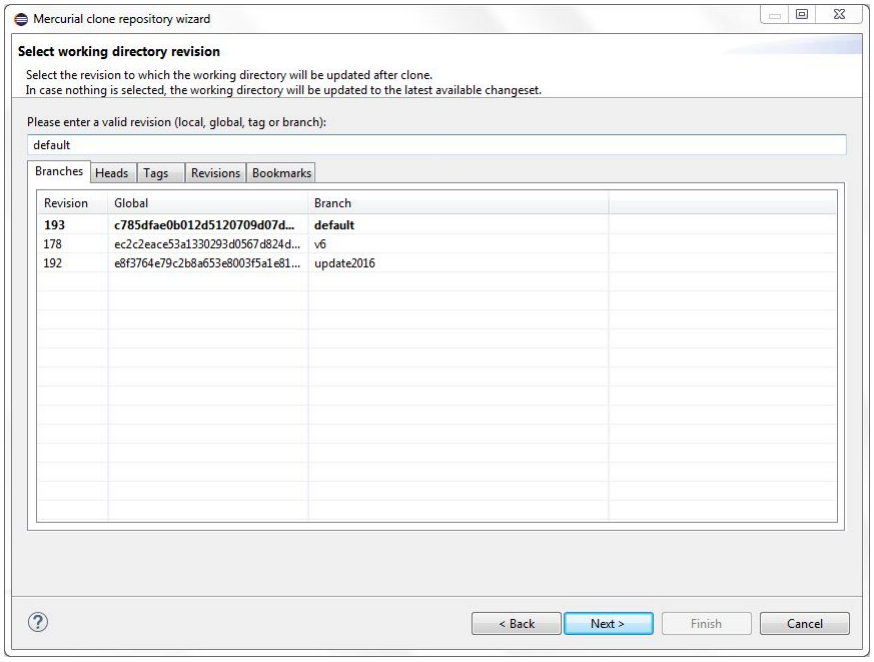

7. In the Import window, select Mercurial and Clone Existing Mercurial Repository, and press Next.



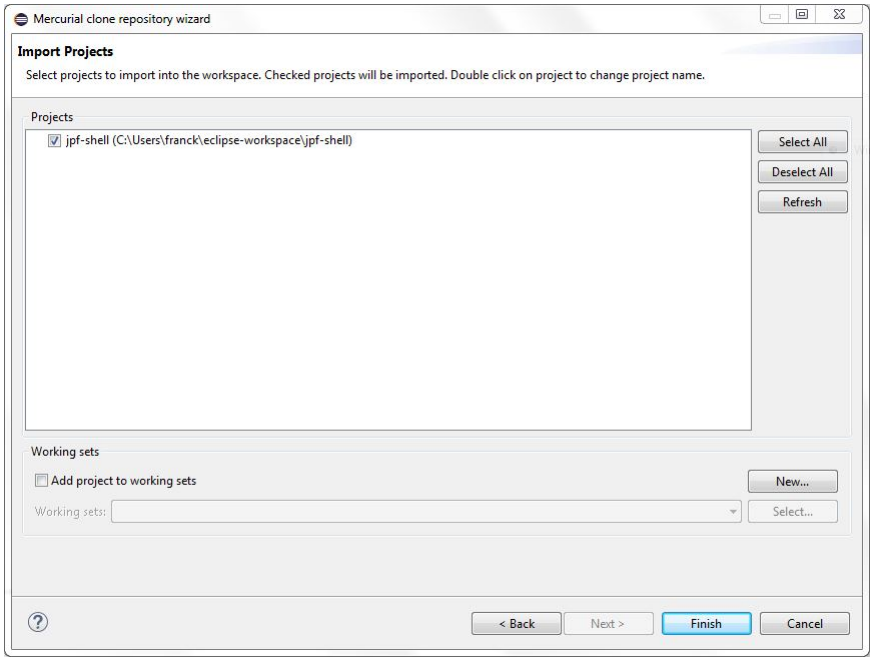

8. In the Clone repository window, enter URL `https://jpf.byu.edu/hg/jpf-shell`, and press Next.




9. In the Select working directory revision, check that the default revision is selected, and press Next.



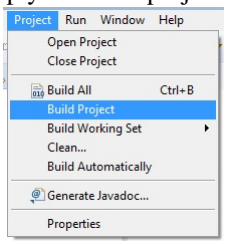

10. In the Import projects window, press Finish.




11. Update the `site.properties` file as described in Section 1.9.3.

### 1.9.6 Updating Sources with Mercurial within Eclipse

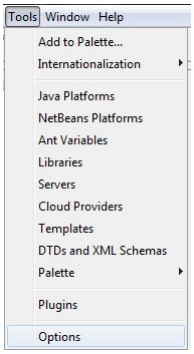
Simply build the project `jpf-shell`: select Project from the menu and then select Build Project.



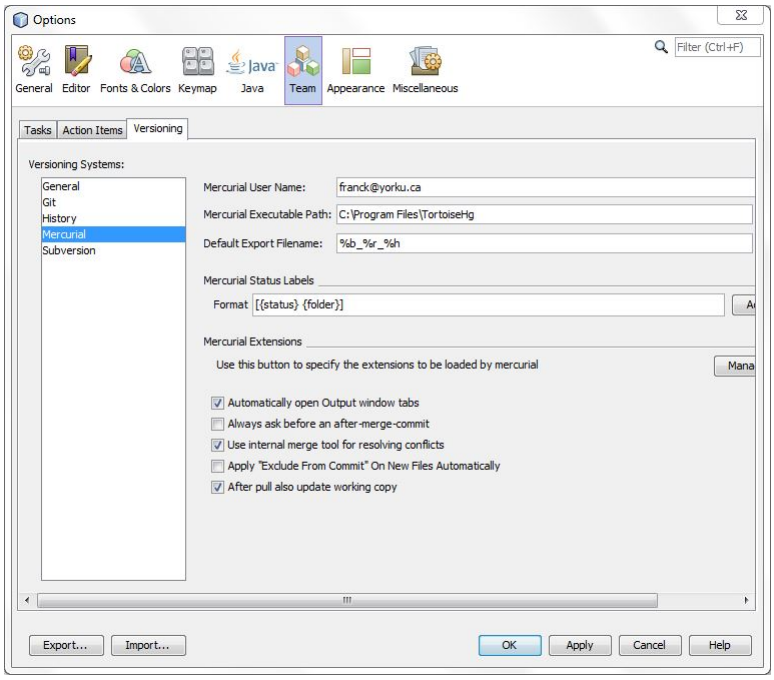

### 1.9.7 Installing Sources with Mercurial within NetBeans

How to install Mercurial is beyond the scope of this book. We refer the reader to www.mercurial-scm.org. We assume that the path to the `hg` command is already part of the system environment variable `PATH` (see Section 1.1.3). To install the JPF extension `jpf-shell` within NetBeans with Mercurial, follow the eight steps below.

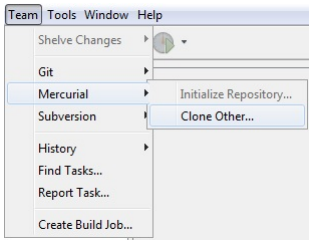
1. In NetBeans, select Tools from the menu item and then select Options.



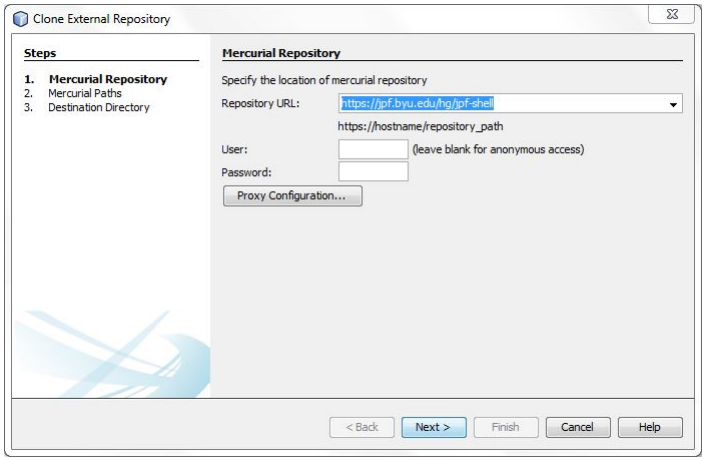
2. In the Options window, select the Team icon and the Versioning tab. Ensure that the Mercurial Executable Path is set correctly. Press OK.



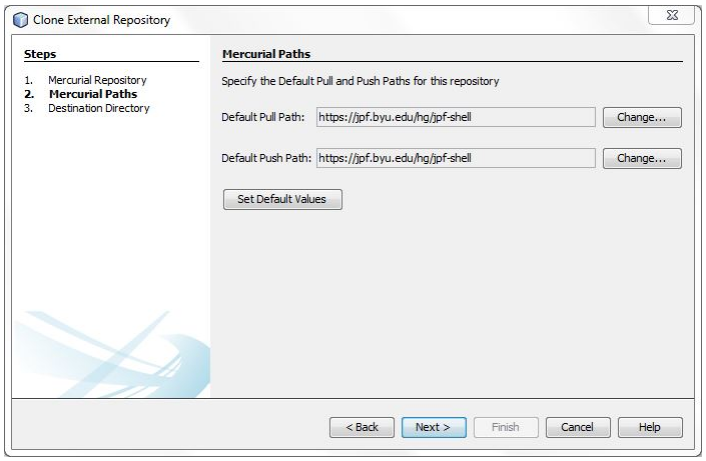

3. In NetBeans, select Team from the menu, then select Mercurial, and finally select Clone Other ...



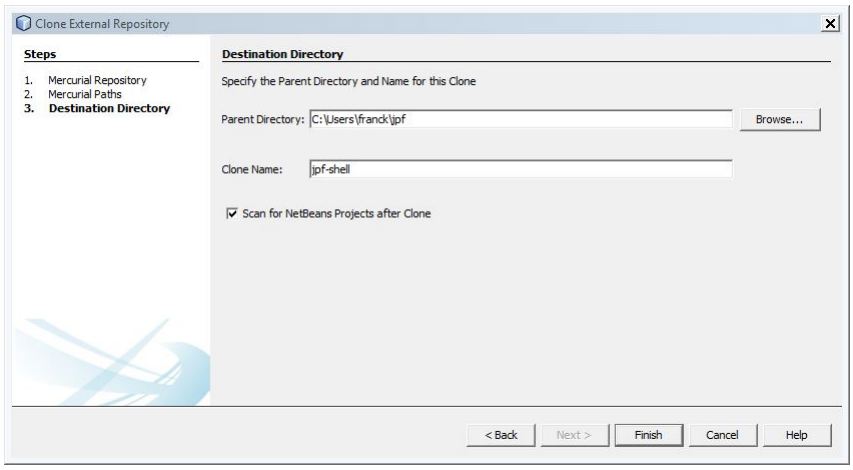

4. In the Clone Extenal Expository window, enter the Repository URL `https://jpf.byu.edu/hg/jpf-shell` and press Next.



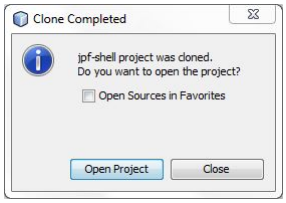

5. In the Clone Extenal Expository window, press Next.




6. In the Clone External Expository window, set the Parent Directory, and press Finish.



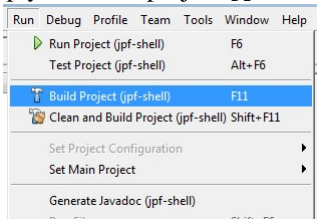

7. In the Clone Completed window, press Open Project.




8. Update the `site.properties` file as described in Section 1.9.3.

### 1.9.8 Updating Sources with Mercurial within NetBeans

Simply build the project `jpf-shell`: select Run from the menu and then select Build Project.



# Chapter 2

# Running JPF

Now that we have discussed how to install JPF, let us focus on how to run JPF. It can be run in several different ways. In Section 2.1 and 2.2 we first show how to run JPF in a shell (Linux and OS X) or command prompt (Windows). How to run JPF within Eclipse and NetBeans are the topics of Section 2.3 and 2.4, respectively.

## 2.1 Running JPF within a Shell or Command Prompt

Let us use the notorious "Hello World" example to show how to run JPF in its most basic form. Consider the following Java application.

```
public class HelloWorld {
  public static void main(String [] args) {
    System.out.println("Hello World!");
  }
}
```

Assume that the file `HelloWorld.class` can be found in the directory `/cs/home/franck/examples`. We create the application properties file named `HelloWorld.jpf`[1] with the following content.

```
target=HelloWorld
classpath=/cs/home/franck/examples
```

The key `target` has the name of the application to be checked by JPF as its value. The key `classpath` has JPF's classpath as its value. It is important not to mix up JPF's classpath with Java's classpath. We will come back to this later.

In case we create the application properties file `HelloWorld.jpf` in the same directory as the file `HelloWorld.class`, it has the following content.

```
target=HelloWorld
classpath=.
```

In this case, JPF's classpath is set to the current directory.

To run JPF on this example, open a shell (Linux and OS X) or command prompt (Windows) in the directory that contains the application properties file `HelloWorld.jpf`, and type

```
jpf HelloWorld.jpf
```

This results in output similar to the following.

[1] Although the name of the application properties file does not have to match the name of the Java application—we could have called it, for example, `Test.jpf`—we will use that convention in this book.

```
1  JavaPathfinder core system v8.0 (rev ...) - (C) 2005-2014 United States Government. All ri
2
3
4  ============================================================ system under test
5  HelloWorld.main()
6
7  ============================================================ search started: 6/20/18 1:25 PM
8  Hello World!
9
10 ============================================================ results
11 no errors detected
12
13 ============================================================ statistics
14 elapsed time:     00:00:00
15 states:           new=1, visited=0, backtracked=1, end=1
16 search:           maxDepth=1, constraints hit=0
17 choice generators: thread=1 (signal=0, lock=1, shared ref=0), data=0
18 heap:             new=366, released=14, max live=0, gc-cycles=1
19 instructions:     4158
20 max memory:       236MB
21 loaded code:      classes=61, methods=1195
22
23 ============================================================ search finished: 6/20/18 1:25 PM
```

Line 1 contains general information. It tells us that we used version 8.0 of JPF. The United States Government holds the copyright of JPF. The remainder of the output is divided into several parts. The number of parts, their headings and content can be configured. The above output is produced by the default configuration. The first part, line 4–5, describes the system under test. In this case, it is the `main` method of the `HelloWorld` class. The second part, line 7–8, contains the output produced by the system under test and the date and time when JPF was started. In this case, the output is `Hello World!` If the output `I won't say it!` is produced instead, the `classpath` has not been set correctly and, as a consequence, JPF checks the `HelloWorld` application which is part of `jpf-core`. The third part, line 10–11, contains the results of the model checking effort by JPF. In this case, no errors were detected. By default, JPF checks for uncaught exceptions and deadlocks. The fourth and final part, line 13–21, contains some statistics. We will discuss them below. The output ends with line 23 which contains the date and time when JPF finished.

It remains to discuss the statistics part. Line 14 describes the amount of time it took JPF to model check the `HelloWorld` application. Since it took less than one second, JPF reports zero hours, zero minutes and zero seconds.

Line 15 categorizes the states visited by JPF. A state is considered *new* the first time it is visited by JPF. If a state is visited again, it is counted as *visited*. The final states are also called *end* states. Those states reached as a result of a backtrack are counted as *backtrack*. In the above example, JPF visits a state which is an end state (1) and subsequently backtracks to the initial state (0) as can be seen in Figure 2.1. We will come back to this classification of states in Chapter 9.

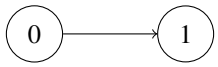



Figure 2.1: State-transition diagram.

Line 16 provides us with some data about the search for bugs by JPF. The search of JPF is similar to the traversal of a directed graph. The states of JPF correspond to the vertices of the graph and the transitions of JPF correspond to the edges of the graph. In a search, the *depth* of a state is the length of the partial execution, a sequence of transitions, along which the state is discovered. From the above diagram, we can conclude that the maximal depth is one in our

example. During the search, JPF checks some *constraints*. By default, it checks two constraints. Firstly, it checks that the depth of the seach is smaller than or equal to the value of the key `search.depth_limit`. By default, its value is $2^{31} - 1$. This JPF property can be configured as we will discuss in Chapter 5. Secondly, it checks that the amount of remaining memory is smaller than or equal to the value of the key `search.min_free`. By default, its value is $2^{20}$. Also this JPF property can be configured. In our example, no constraints are violated and, hence, the number of constraint hits is zero.

Line 17 contains information about the choice generators. These capture the nondeterministic choices, either caused by randomization or concurrency, and will be discussed in more detail in Chapter **??**. Some statistics about the heap of JPF's virtual machine are given in line 18.

Line 19 specifies the number of bytecode instructions that have been checked by JPF. The maximum amount of memory used by JPF is given in line 20. Line 21 contains the number of classes and methods that have been checked by JPF.

If the class `HelloWorld` were part of the package `test` and the file `HelloWorld.class` could be found in the directory `/cs/home/franck/examples/test`, then the application properties file would contain the following.

```
target=test.HelloWorld
classpath=/cs/home/franck/examples
```

## 2.2 Detecting Bugs with JPF

Let us now present some examples of JPF detecting a bug. The examples are kept as simple as possible. As a consequence, they are not realistic representatives of applications on which one might want to apply JPF. However, they allow us to focus on detecting bugs with JPF.

Recall that JPF checks for uncaught exceptions and deadlock by default. Consider the following system under test.

```
1 public class UncaughtException {
2   public static void main(String [] args) {
3     System.out.println(1 / 0);
4   }
5 }
```

Obviously, line 3 throws an exception that is not caught. Running JPF on this example results in output similar to the following.

```
1  JavaPathfinder core system v8.0 (rev ...) - (C) 2005-2014 United States Government. All ri
2
3
4  ============================================================ system under test
5  UncaughtException.main()
6
7  ============================================================ search started: 08/07/18 7:05 PM
8
9  ============================================================ error 1
10 gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
11 java.lang.ArithmeticException: division by zero
12 at UncaughtException.main(UncaughtException.java:3)
13
14
15 ============================================================ snapshot #1
16 thread java.lang.Thread:{id:0,name:main,status:RUNNING,priority:5,
17 lockCount:0,suspendCount:0}
18   call stack:
```

```
19   at UncaughtException.main(UncaughtException.java:3)
20
21
22 ====================================================== results
23 error #1: gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
24 "java.lang.ArithmeticException: division by zero a..."
25
26 ====================================================== statistics
27 elapsed time:  00:00:00
28 states:        new=1, visited=0, backtracked=0, end=0
29 search:        maxDepth=1, constraints hit=0
30 choice generators: thread=1 (signal=0, lock=1, shared ref=0), data=0
31 heap:          new=380, released=0, max live=0, gc-cycles=0
32 instructions:  3312
33 max memory:    90MB
34 loaded code:   classes=65, methods=1226
35
36 ====================================================== search finished: 08/07/18 7:05 PM
```

Line 9–12 report the bug detected. JPF can be configured to detect multiple errors, as we will discuss in Chapter 5. By default, JPF finishes after detecting the first bug. Line 10 describes the type of bug detected. In this case, the `NoUncaughtExceptionProperty` is violated and, hence, an exception has not been caught. Line 11 and 12 provide the stack trace. From this stack trace we can deduce that the uncaught exception is an `ArithmeticException` and it occurs in line 3 of the `main` method of the `UncaughtException` class. Line 15–19 provides some information for each relevant thread. In this case, there is only a single thread. For each thread JPF records a unique identifier, its name, its status, its priority and two counters. Furthermore, it prints the stack trace of each relevant thread. Line 22-24 summarize the results.

If an assertion, specified by the `assert` statement, fails, an `AssertionError` is thrown. Hence, JPF can detect these. Consider the following application.

```
1 public class FailingAssertion {
2   public static void main(String[] args) {
3     int i = 0;
4     assert i == 1;
5   }
6 }
```

The output produced by JPF for this example is very similar to that produced for the previous example. JPF reports that an uncaught `AssertionError` occurs in line 4 of the `main` method of the `FailingAssertion` class.

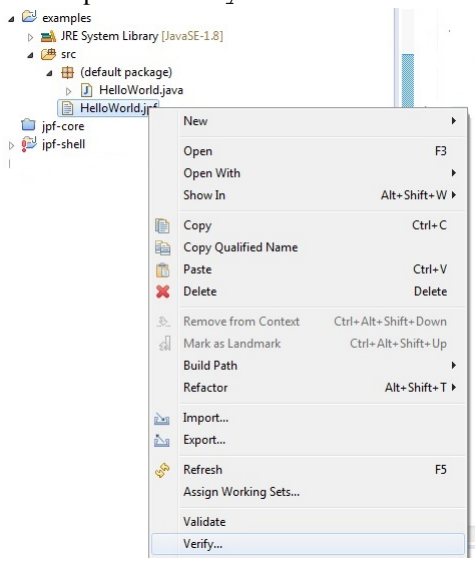
## 2.3 Running JPF within Eclipse

We assume that the reader has installed the JPF plugin (see Section 1.5). Let us also assume that we have created an Eclipse project named `examples` which contains the class `HelloWorld` in the default package. If we installed JPF as described in Section 1.3, then the file `HelloWorld.java` can be found in the directory `/cs/home/franck/workspace/examples/src` (Linux and OS X) and `C:\Users\franck\workspace\examples\src` (Windows). The corresponding file `HelloWorld.class` can be found in the directory `/cs/home/franck/workspace/examples/bin` (Linux and OS X) and `C:\Users\franck \workspace\examples\bin` (Windows).

Next, we create the `HelloWorld.jpf` file. Although this file can be placed in any directory, it is most convenient to place it in the same directory as the `HelloWorld.java` file. As before, the most basic application properties file only contains two keys: `target` and `classpath`. In this case, the value of `classpath` is the directory that contains `HelloWorld.class`. For example, for Windows the content of `HelloWorld.jpf` becomes

```
target=HelloWorld
classpath=C:/Users/franck/workspace/examples/bin
```

Finally, to run JPF on this example within Eclipse, right click on `HelloWorld.jpf` in the package explorer and select the option `Verify...`




It results in the output similar to what we have seen in the previous section, preceded by something like

```
Executing command: java -jar C:\Users\franck\jpf\jpf-core\build\RunJPF.jar
+shell.port=4242 C:\Users\franck\workspace\examples\src\HelloWorld.jpf
```

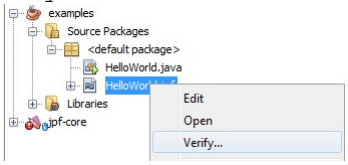
## 2.4 Running JPF within NetBeans

We assume that the reader has installed the JPF plugin (see Section 1.8). Let us also assume that we have created a NetBeans project named `examples` which contains the class `HelloWorld` in the default package. If we installed JPF as described in Section 1.6, then the file `HelloWorld.java` can be found in the directory `/cs/home/franck/NetBeansProjects/examples/src` (Linux and OS X) and `C:\Users\franck\NetBeansProjects\examples\src` (Windows). The corresponding file `HelloWorld.class` can be found in the directory `/cs/home/franck/NetBeansProjects/examples/build/classes` (Linux and OS X) and `C:\Users\franck\NetBeansProjects\examples\build\classes` (Windows).

Next, we create the `HelloWorld.jpf` file. Although this file can be placed in any directory, it is most convenient to place it in the same directory as the `HelloWorld.java` file. As before, the most basic application properties file only contains two keys: `target` and `classpath`. In this case, the value of `classpath` is the directory that contains `HelloWorld.class`. For example, for Windows the content of `HelloWorld.jpf` becomes

```
target=HelloWorld
classpath=C:/Users/franck/NetBeansProjects/examples/build/classes
```

Finally, to run JPF on this example within NetBeans, right click on `HelloWorld.jpf` and select the option `Verify...`




It results in the output similar to what we have seen in the previous section, preceded by something like

```
Executing command: java -jar C:\Users\franck\jpf\jpf-core\build\RunJPF.jar
+shell.port=4242 C:\Users\franck\NetBeansProjects\examples\src\HelloWorld.jpf
```

# Chapter 3

# Configuring JPF

Since JPF has been designed so that it can be easily extended, it should come as no surprise that little is "hard wired" in JPF. As a consequence, there is a need for a mechanism that allows us to configure JPF. How to configure JPF is the topic of this chapter.

## 3.1 Properties Files

JPF can be configured by command line arguments and properties files. We focus here only on the latter. A property file defines properties. Each property consists of a key and a (string) value, separated by an = sign. For example, `target=HelloWorld` assigns to the key `target` the value `HelloWorld`. We distinguish between three different types of properties file:

1. the site properties file,
2. the projects properties files, and
3. the applications properties files.

In general, these files are loaded in the above order. To determine the exact order, one can use the command line argument `-log`. For example, typing `jpf -log HelloWorld.jpf` in a shell or command prompt produces the output as seen in Section 2.1 preceeded by output similar to the following.

```
loading property file: /cs/home/franck/.jpf/site.properties
loading property file: /cs/home/franck/jpf/jpf-core/jpf.properties
loading property file: HelloWorld.jpf
collected native_classpath=/cs/home/franck/jpf/jpf-core/build/jpf.jar,
/cs/home/franck/jpf/jpf-core/build/jpf-annotations.jar,
collected native_libraries=null
```

First, the site properties file is loaded. After that, the properties file of `jpf-core` is loaded. And finally, the application properties file `HelloWorld.jpf` is loaded. We will discuss the `native_classpath` and `native_libraries` later in this chapter. Next, we will discuss each type of properties file in some detail.

### 3.1.1 The Site Properties File

This file is named `site.properties`. In Section 1.1.4 we already discussed how to create this file. It contains the key `extensions`, whose value is the directory where `jpf-core` can be found. For example, `site.properties` may look like

```
# JPF site configuration
jpf-core=${user.home}/jpf/jpf-core
extensions=${jpf-core}
```

Note that the properties file contains two keys: `jpf-core` and `extensions`. In the above file, `${jpf-core}` represents the value associated with the key `jpf-core`, that is, it is the directory in which `jpf-core` can be found.

### 3.1.2 The Project Properties Files

Each project, such as `jpf-core` and `jpf-shell`, has its own properties file. This file is named `jpf.properties`. The file can be found in the root directory of the project. This file is not only used to configure the project. It is also used to build the project.

The first entry of `jpf.properties` consists of the project's name as key and `${config_path}` as value. The value of the key `config_path` is the directory of the `jpf.properties` file. For example, the `jpf.properties` file of the project `jpf-shell` starts with

```
jpf-shell=${config_path}
```

JPF is implemented as a Java virtual machine (JVM). Therefore, JPF has a classpath. JPF's classpath should contain the Java bytecode of the classes that need to be model checked. That is, it should contain the bytecode of the system under test and any classes used by it.

Since JPF is implemented in Java, it runs on top of a JVM, which we will call the host JVM. This host JVM has a classpath as well. This classpath should contain the bytecode of the classes that are needed to run JPF and its extensions. To distinguish this classpath from JPF's classpath, we call it the native classpath. Both classpaths may contain directories and jar files.

In general, each project adds the examples it includes to the classpath. For example, `jpf-core` includes `HelloWorld` as an example. This class is part of the classpath of `jpf-core`. It also adds those classes needed to run the project to the native classpath. For example, the jar file `/cs/home/franck/jpf/jpf-shell/build jpf-shell.jar`, which contains Java classes that are needed to run `jpf-shell`, is part of the native classpath of `jpf-shell`.

Each project has its own classpath and native classpath. To distinguish them, they are prefixed by the project's name. For example, the classpath of `jpf-core` is named `jpf-core.classpath` and the native classpath of `jpf-shell` is named `jpf-shell.native_classpath`. Furthermore, each project may also have its own sourcepath. This path contains the Java source code of the classes that are model checked. Although JPF checks Java bytecode, it uses the Java source code to generate feedback to the user when it detects a bug. In particular, it refers to the line numbers of the source code, which makes it easier for the user to locate the bug. For example, the directory `/cs/home/franck/jpf/jpf-core/src/examples`, which contains the Java source code of the examples of the `jpf-core` project, is part of the source path of `jpf-core`. How the classpaths, the native classpaths and the source paths are combined will be discussed later in this chapter.

The `jpf.properties` file of the `jpf-core` contains the following.

```
jpf-core.classpath=\
  ${jpf-core}/build/jpf-classes.jar;\
  ${jpf-core}/build/examples

jpf-core.native_classpath=\
  ${jpf-core}/build/jpf.jar;\
  ${jpf-core}/build/jpf-annotations.jar

jpf-core.sourcepath=\
  ${jpf-core}/src/examples
```

The classpath of `jpf-core` contains the jar file `${jpf-core}/build/jpf-classes.jar`. This file contains the bytecode of the classes that model some classes of the Java standard library such as `java.io.File`. These classes are known as peer classes. When model checking an application that uses any of these classes, JPF checks

the peer class rather than the original class of the Java standard library. We will discuss peer classes in more detail in Chapter 10. The classpath of `jpf-core` also contains the directory `${jpf-core}/build/examples` which contains the bytecode of the examples of `jpf-core`.

The native classpath of `jpf-core` consists of two jar files. The file `${jpf-core}/build/jpf.jar` contains the bytecode of the classes that make up the core of JPF. The file `${jpf-core}/build/annotations.jar` contains the bytecode of JPF's annotation classes.

The source path of `jpf-core` solely consists of the directory `${jpf-core}/src/examples` which contains the source code of the examples of `jpf-core`.

The `jpf.properties` file of `jpf-core` defines more than one hundred other properties. Some of them we will discuss in Section 3.2. Many others we will encounter in the remainder of this book.

### 3.1.3 The Application Properties Files

As we have already seen in Chapter 2, to run jpf we have to write an application properties file. In this file we need to provide values for the keys `target` and `classpath`. The former is the name of the system under test and the latter is part of JPF's classpath. In the application properties file we can set other properties as well.

## 3.2 JPF Properties

There are a large number of JPF properties that can be set. To determine which JPF properties have been set, one can use the `-show` command line argument. For example, typing `jpf -show HelloWorld.jpf` in a shell or command prompt produces the output as seen in Section 2.1 preceded by output whose beginning and end are similar to the following.

```
----------- Config contents
branch_start = 1
cg.boolean.false_first = true
cg.break_single_choice = false
...
vm.tree_output = true
vm.untracked = true
vm.verify.ignore_path = true
```

### 3.2.1 Command Line Arguments

Consider the following app that takes two command line arguments.

```
public class CommandLineArguments {
  public static void main(String[] args) {
    System.out.println(args[0] + " " + args[1] + "!");
  }
}
```

To run JPF on the above app with the command line arguments `Hello` and `World` one can uses the following application properties file.

```
target = CommandLineArguments
target.args = Hello,World
```

The property `target.args` captures the command line arguments that are used for the verification effort. Instead of a comma, the command line arguments can also be separated by a semicolon. Note that apart from the comma, the command line arguments are not separated by any white space. If we were to introduce such whitespace, it would be considered part of the command line arguments.

### 3.2.2 Randomization

Consider the following app.

```
import java.util.Random;

public class Choice {
  public static void main(String[] args) {
    Random random = new Random();
    if (random.nextBoolean()) {
      System.out.println("if");
    } else {
      System.out.println("else");
    }
  }
}
```

If we only set the properties `target` and `classpath` to the appropriate values, JPF produces the following output.

```
JavaPathfinder core system v8.0 (rev ...) - (C) 2005-2014 United States Government. All rig


============================================================ system under test
Choice.main()

============================================================ search started: 11/30/18 8:22 AM
if

============================================================ results
no errors detected
...
```

Although the above app has two potential executions, one printing "if" and the other printing "else," JPF only executes one of them, namely printing "if," as reflected by the above output. If we run JPF multiple times, it will take the if branch every time.

#### The `cg.enumerate_random` Property

In order the unleash the power of JPF, we have to set the property `cg.enumerate_random` to true (its default value is false). In that case, JPF checks both branches as illustrated by the following output.

```
JavaPathfinder core system v8.0 (rev ...) - (C) 2005-2014 United States Government. All rig


============================================================ system under test
Choice.main()

============================================================ search started: 11/30/18 8:56 AM
else
if

============================================================ results
no errors detected
...
```

The prefix `cg` stands for choice generator. A choice generator represents a nondeterministic choice in the code. Such a choice, be it the result of randomization or concurrency, is captured by the class `ChoiceGenerator` of the package `gov.nasa.jpf.vm`. We will return to this class later in this book. For now, we focus on the properties related to choice generators.

### The `cg.boolean.false_first` Property

By default, the value of the property `cg.boolean.false_first` is true, that is, if `cg.enumerate_random` is set to true and a random boolean is chosen in the code, then JPF considers the false value first. That explains why else appears before if in the above output. If we set `cg.boolean.false_first` to false, then JPF produces the following output.

```
JavaPathfinder core system v8.0 (rev ...) - (C) 2005-2014 United States Government. All ri


====================================================== system under test
Choice.main()

====================================================== search started: 11/30/18 9:35 AM
if
else

====================================================== results
no errors detected
...
```

### The `cg.randomize_choices` and `cg.seed` Properties

Consider the following app.

```
import java.util.Random;

public class Choice {
  public static void main(String[] args) {
    Random random = new Random();
    final int ALTERNATIVES = 3;
    switch (random.nextInt(ALTERNATIVES)) {
      case 0 :
        System.out.println("0");
        break;
      case 1 :
        System.out.println("1");
        break;
      case 2 :
        System.out.println("2");
        break;
    }
  }
}
```

If the property `cg.enumerate_random` is set to true, then JPF produces the following output for the above app.

```
JavaPathfinder core system v8.0 (rev ...) - (C) 2005-2014 United States Government. All ri
```

```
====================================================== system under test
Choice.main()

====================================================== search started: 11/30/18 12:19 PM
0
1
2

====================================================== results
no errors detected
...
```

The order in which the alternatives related to a random choice are considered by JPF can be customized by the using the property `cg.randomize_choices`. This property can take three different values. By default, this property has the value `NONE`, that is, the alternatives are not randomized. If we set the property `cg.randomize_choices` to `VAR_SEED` and the property `cg.enumerate_random` true then the alternatives are randomized. That is, JPF randomly determines the order in which the alternatives are considered. For the above example, JPF produces a random permutation of 0, 1 and 2. The third possible value for the property `cg.randomize_choices` is `FIXED_SEED`. In this case, JPF determines the order in which the alternatives are considered based on a fixed seed. This seed is captured by the property `cg.seed`. Its default value is 42. In that case, JPF gives rise to the permutation 1, 0, 2. If we set the property `cg.seed` to 41, then JPF produces the following output.

```
JavaPathfinder core system v8.0 (rev ...) - (C) 2005-2014 United States Government. All ri


====================================================== system under test
Choice.main()

====================================================== search started: 11/30/18 12:26 PM
0
2
1

====================================================== results
no errors detected
...
```

### The `cg.break_single_choice` Property

Now consider the following app.

```
1 import java.util.Random;
2
3 public class Choice {
4   public static void main(String[] args) {
5     Random random = new Random();
6     random.nextInt(1);
7   }
8 }
```

Note that the choice in line 6 only has a single branch. If we run JPF for the above app, it produces the following statistics.

```
1 ========================================================== statistics
2 elapsed time:  00:00:00
3 states:        new=1,visited=0,backtracked=1,end=1
4 search:        maxDepth=1,constraints=0
5 choice generators: thread=1 (signal=0,lock=1,sharedRef=0,threadApi=0,reschedule=0), data=0
6 heap:          new=354,released=12,maxLive=0,gcCycles=1
7 instructions:  3182
8 max memory:    240MB
9 loaded code:   classes=62,methods=1341
```

In line 5, the number of created choice generators is reported. A distinction is made between those related to concurrency (`thread`) and randomization (`data`). We will get to the former in Chapter **??**. Note that, by default, a choice with a single branch does not give rise to the creation of a choice generator, since there are zero data choice generators created. However, if we set the property `cg.break_single_choice` to true (its default value is false), then JPF produces the following statistics for the above app.

```
========================================================== statistics
elapsed time:  00:00:00
states:        new=27,visited=0,backtracked=27,end=1
search:        maxDepth=27,constraints=0
choice generators: thread=26 (signal=0,lock=26,sharedRef=0,threadApi=0,reschedule=0), data=
heap:          new=354,released=12,maxLive=352,gcCycles=25
instructions:  3208
max memory:    240MB
loaded code:   classes=62,methods=1341
```

In this case, one data choice generator has been created, which corresponds to the choice in line 6 of the above app.

## 3.3 Using an Extension of JPF

To use an extension, we set the `@using` property in the application properties file. For example, to use the `jpf-shell` extension when checking the `HelloWorld` app, we set the `@using` property as follows.

```
@using=jpf-shell

target=HelloWorld
classpath=.
```

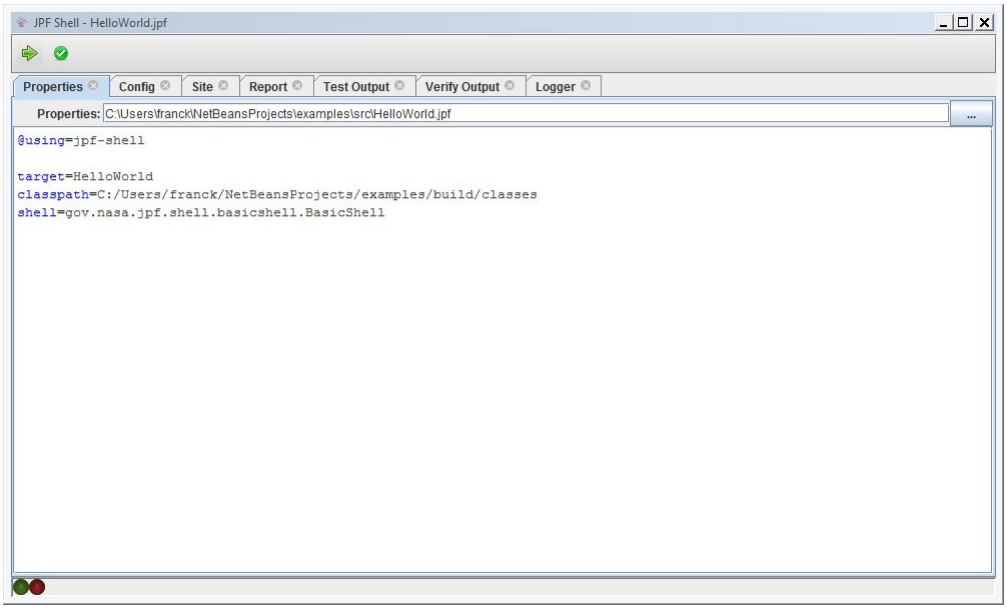
If we run JPF with the `-show` option, then a number of new properties and their values, including the following, are output.

```
...
shell.commands = test,verify
shell.commands.test = gov.nasa.jpf.shell.commands.TestCommand
shell.commands.verify = gov.nasa.jpf.shell.commands.VerifyCommand
shell.panels = properties,config,site,report,test,verify,logging
shell.panels.config = gov.nasa.jpf.shell.panels.ConfigPanel
shell.panels.logging = gov.nasa.jpf.shell.panels.LoggingPanel
shell.panels.properties = gov.nasa.jpf.shell.panels.PropertiesPanel
shell.panels.report = gov.nasa.jpf.shell.panels.ReportPanel
shell.panels.site = gov.nasa.jpf.shell.panels.SitePanel
shell.panels.test = gov.nasa.jpf.shell.panels.TestConsolePanel
shell.panels.verify = gov.nasa.jpf.shell.panels.VerifyConsolePanel
...
```

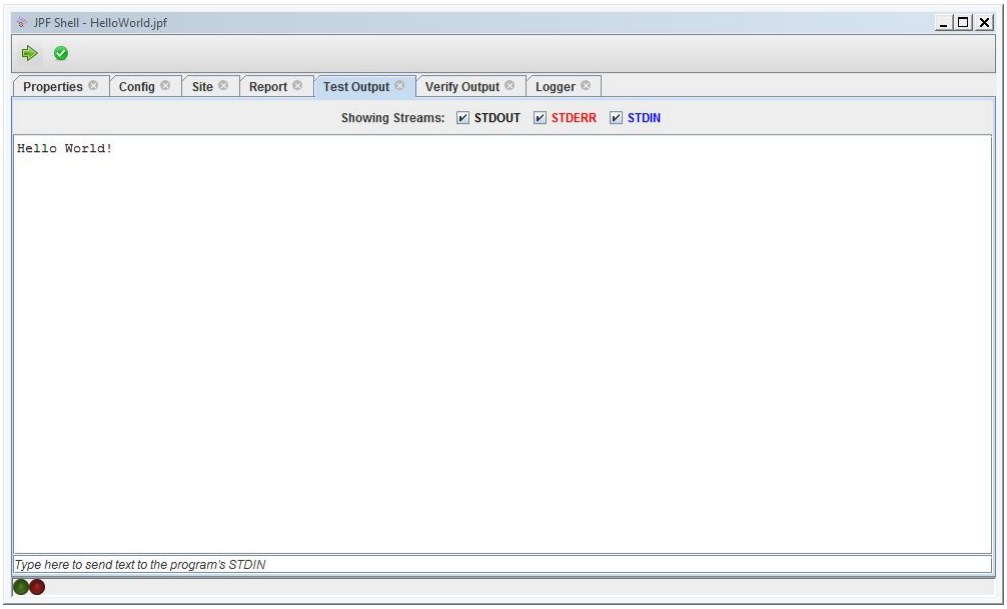
To use the `jpf-shell` extension, one has to set the `shell` property as follows.

```
@using=jpf-shell

target=HelloWorld
classpath=.

shell=gov.nasa.jpf.shell.basicshell.BasicShell
```

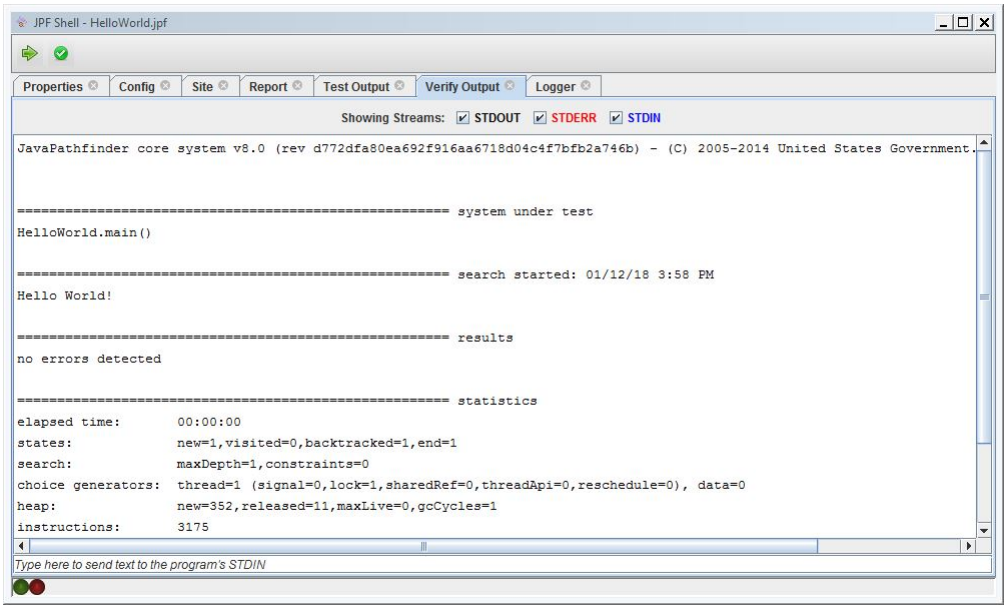
When we run JPF with the above application properties file, the window below appears.



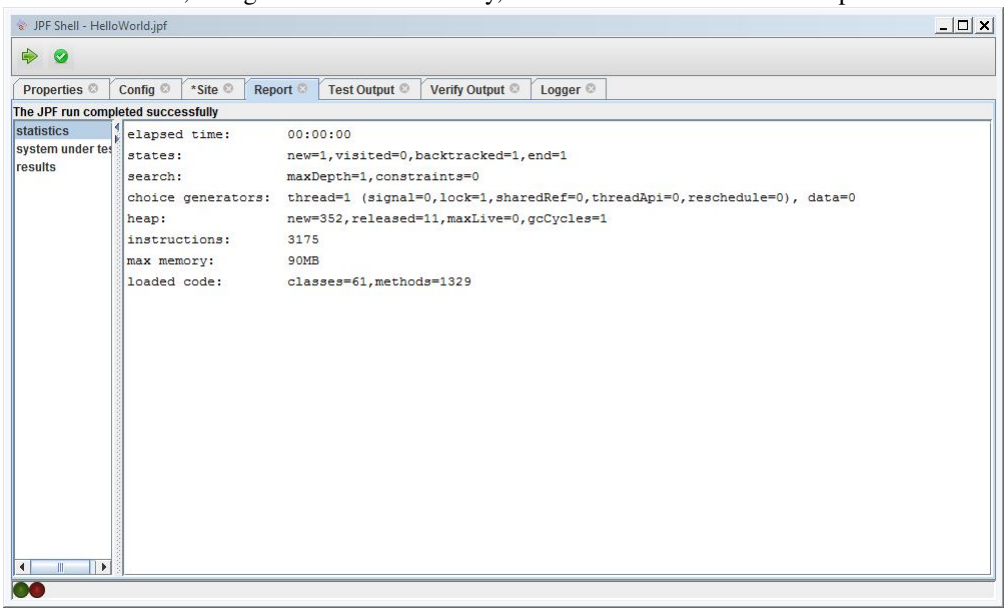

If we click the arrow button, then the app is tested, that is, its code is run. If we click the check mark button, then the app verified, that is, JPF is run. The output of the former can be found under the Test Output tab.




The output of the latter can be found under the Verify Output tab.




This information, though formatted differently, can also be found under the Report tab.

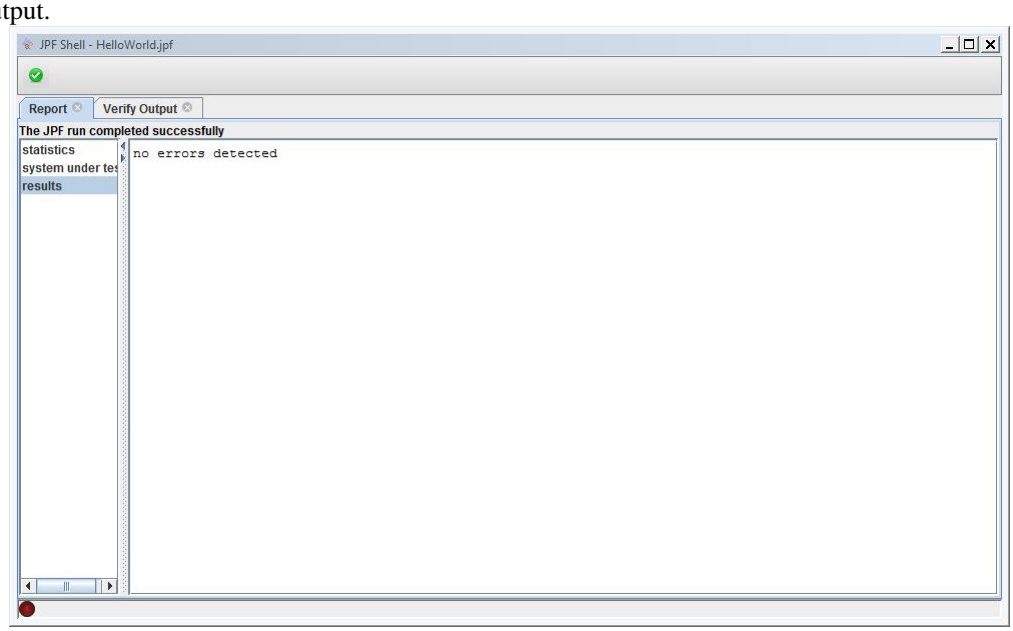



The window can be configured by setting properties. For example, consider the following application properties file.

```
@using=jpf-shell

target=HelloWorld
classpath=.

shell=gov.nasa.jpf.shell.basicshell.BasicShell
shell.commands=verify
shell.panels=report,verify
```

We only specify a single command, namely `verify`. As a consequence, the window only contains the check mark button. As panels, we select `report` and `verify`. As a result, the window has only two tabs: Report and Verify

Output.




## 3.4 Paths

As we already mentioned before, each project may define its own classpath, native classpath, and source path. These paths can also be defined in the application properties file. Next, we will discuss how these are combined.

Assume that the extensions `jpf-numeric` and `jpf-shell` have both been installed. Consider the following application properties file.

```
@using=jpf-numeric
@using=jpf-shell

target=Test
classpath=/cs/home/franck/examples
```

Assume that the project property files of the extensions `jpf-numeric` and `jpf-shell` define the properties `jpf-numeric.classpath` and `jpf-shell.classpath`, respectively. When JPF is run with the above application properties file, to find the bytecode of the `Test` app, it first looks in `/cs/home/franck/examples`. If it cannot find it there, then it looks in the directories and jar files assigned to `jpf-shell.classpath`. If it cannot find it there either, then it looks in the directories and jar files assigned to `jpf-numeric.classpath`. If it cannot find it there either, then it looks in the directories and jar files assigned to `jpf-core.classpath`. If it cannot find it there either, then JPF produces the following output.

```
[SEVERE] cannot load application class Test
```

Hence, JPF's classpath consists of a combination of all values of the properties that end in `.classpath`. Similarly, the native classpath and source path are combined.

# Chapter 4

# Using a Listener

To extract information from JPF during its verification effort, JPF uses event driven programming. We assume that reader is already familiar with this programming paradigm. In JPF, events signal, for example, that a bytecode instruction has been executed, that a class has been loaded, or that the garbage collector has started. To handle these events, JPF contains a number of listeners. Such a listener is registered with JPF by setting the value of the key `listener` to the class implementing the listener in the application properties file. For example, extracting a representation of the state space in the DOT format by means of the `StateSpaceDot` listener, which is part of the package `gov.nasa.jpf.listener`, can be specified in the application properties file as follows.

```
listener=gov.nasa.jpf.listener.StateSpaceDot
```

The package `gov.nasa.jpf.listener` contains a number of listeners. Extensions of JPF, such as `jpf-shell`, contain listeners as well. In the remainder of this chapter we discuss some of them.

## 4.1 Using the `StateSpaceDot` Listener

As mentioned above, the `StateSpaceDot` listener extracts a representation of the state space in DOT format. Consider the application

```
import java.util.Random;

public class StateSpace {
  public static void main(String[] args) {
    Random random = new Random();
    System.out.print("0 ");
    if (!random.nextBoolean()) {
      System.out.print("1 ");
      if (!random.nextBoolean()) {
        System.out.print("3 ");
      } else {
        System.out.print("4 ");
        if (!random.nextBoolean()) {
          System.out.print("5 ");
        } else {
          System.out.print("6 ");
        }
      }
    } else {
      System.out.print("2 ");
```

```
    }
  }
}
```

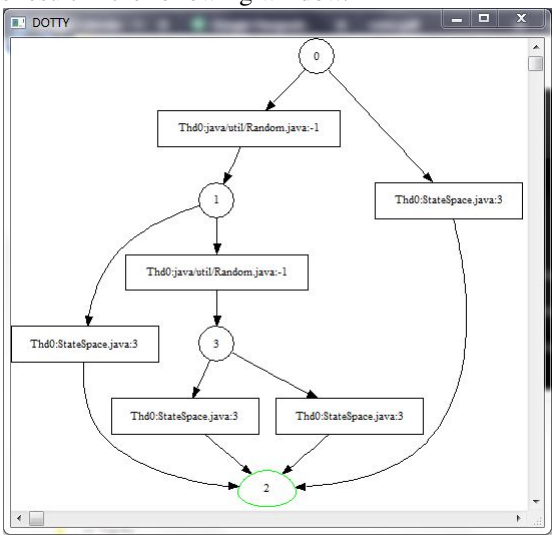
and the application properties file

```
target=StateSpace
classpath=.
listener=gov.nasa.jpf.listener.StateSpaceDot
```

When JPF is run with the above application properties file, it produces a file named `jpf-state-space.dot` and also the usual output. This file can be viewed with the visualization software called Graphviz. More information about Grapviz can be found at www.graphviz.org. We assume that the path to the `dotty` command is already part of the system environment variable `PATH` (see Section 1.1.3). To view the generated file, type

```
dotty jpf-state-space.dot
```

This result in the following window.




In Section 7.4 we will develop a similar listener.

## 4.2 Using the `BudgetChecker` Listener

The listener `BudgetChecker`, which is part of the package `gov.nasa.jpf.listener`, allows us to set constraints for JPF. For example, one can set the maximum number of milliseconds used by JPF. If JPF takes more time than this maximum when using this listener, JPF will terminate and will report that it took more than the set maximum. Apart from the amount time used, we can also constrain the amount of heap space used, the number of instructions executed, the depth of the search and the number of new states. To specify these constraints, the listener has several properties that can be set in the application properties file, as we will show below.

To illustrate the `BudgetChecker` listener, we consider the following app.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class Constraints {
  public static void main(String[] args) {
    Random random = new Random();
    List<Integer> list = new ArrayList<Integer>();
```

```
    for (int i = 0; random.nextBoolean(); i++) {
      list.add(new Integer(i));
    }
  }
}
```

We can run the listener `BudgetChecker` without setting any of its properties and, hence, not setting any constraints, as shown in the application properties file below.

```
target=Constraints
classpath=.
cg.enumerate_random=true
listener=gov.nasa.jpf.listener.BudgetChecker
```

In this case, no constraints are set. As a result, JPF will run out of memory producing the following output

```
JavaPathfinder core system v8.0 (rev ...) - (C) 2005-2014 United States Government. All rig


====================================================== system under test
Constraints.main()

====================================================== search started: 1/27/17 1:01 PM
[SEVERE] JPF out of memory
...
```

We can constrain the maximum amount time used by JPF to 1000 milliseconds as follows. We can assign any long value to the property `budget.max_time`.

```
target=Constraints
classpath=.
cg.enumerate_random=true
listener=gov.nasa.jpf.listener.BudgetChecker
budget.max_time=1000
```

Since JPF cannot complete within one second, it stops after one second and produces the following output.

```
JavaPathfinder core system v8.0 (rev ...) - (C) 2005-2014 United States Government. All rig


====================================================== system under test
Constraint.main()

====================================================== search started: 1/27/17 1:22 PM

====================================================== search constraint
max time exceeded: 00:00:01 >= 00:00:01
...
```

The amount of heap space used by JPF can be constrained to a maximum number of bits as follows. We can assign any long value to the property `budget.max_heap`.

```
target=Constraints
classpath=.
cg.enumerate_random=true
listener=gov.nasa.jpf.listener.BudgetChecker
budget.max_heap=10000000
```

Note that 10000000 bits = 10000000 / (1024 * 1024) megabytes = 9.5 megabytes. Since JPF needs more heap space, it stops and produces the following output.

```
JavaPathfinder core system v8.0 (rev ...) - (C) 2005-2014 United States Government. All rig


====================================================== system under test
Constraints.main()

====================================================== search started: 1/27/17 1:36 PM

====================================================== search constraint
max heap exceeded: 10MB >= 9MB
...
```

One can constrain JPF from checking more than 100 bytecode instructions as follows. We can assign any long value to the property `budget.max_insn`.

```
target=Constraints
classpath=.
cg.enumerate_random=true
listener=gov.nasa.jpf.listener.BudgetChecker
budget.max_insn=100
```

Since this constraint is violated, JPF produces the following output.

```
JavaPathfinder core system v8.0 (rev ...) - (C) 2005-2014 United States Government. All rig


====================================================== system under test
Constraints.main()

====================================================== search started: 1/27/17 1:47 PM

====================================================== search constraint
max instruction count exceeded: 100
...
```

We can also limit the number of states that JPF explores by setting the property `budget.max_state`. We can assign any int value to this property.

```
target=Constraints
classpath=.
cg.enumerate_random=true
listener=gov.nasa.jpf.listener.BudgetChecker
budget.max_state=100
```

Since the statespace of this app consists of more than 100 states, JPF stops and produces the following output.

```
JavaPathfinder core system v8.0 (rev ...) - (C) 2005-2014 United States Government. All rig


====================================================== system under test
Constraints.main()

====================================================== search started: 1/27/17 1:48 PM
```

```
====================================================== search constraint
max states exceeded: 100
...
```

We can also constrain the depth of the search. The corresponding property, `budget.max_depth`, can take any int value.

```
target=Constraints
classpath=.
cg.enumerate_random=true
listener=gov.nasa.jpf.listener.BudgetChecker
budget.max_depth=100
```

Since the depth of the search is greater than 100 for this app, JPF stops and produces the following output.

```
JavaPathfinder core system v8.0 (rev ...) - (C) 2005-2014 United States Government. All ri


====================================================== system under test
Constraints.main()

====================================================== search started: 1/27/17 1:53 PM

====================================================== search constraint
max search depth exceeded: 100
...
```

## 4.3 Using the `EndlessLoopDetector` Listener

The listener `EndlessLoopDetector`, which is part of the package `gov.nasa.jpf.listener`, tries to detect nonterminating loops. Since termination and, hence, nontermination, is undecidable, this listener uses heuristics. As a result, as we will see below, one has to be careful with interpreting the reports it produces.

Consider the following app.

```
public class Loop {
  public static void main(String[] args) {
    while (true) {}
  }
}
```

If we run JPF with the application properties file

```
target=Loop
classpath=.
listener=gov.nasa.jpf.listener.EndlessLoopDetector
```

then it produces the following output.

```
JavaPathfinder core system v8.0 (rev ...) - (C) 2005-2014 United States Government. All ri


====================================================== system under test
Loop.main()

====================================================== search started: 11/26/18 3:58 PM
[WARNING] breaks transition on suspicious loop in thread: main
```

```
      at Loop.main(Loop.java:3)
[WARNING] breaks transition on suspicious loop in thread: main
      at Loop.main(Loop.java:3)

====================================================== results
error #1: gov.nasa.jpf.listener.EndlessLoopDetector
...
```

In the above example, we considered a while loop. The listener can also detect nonterminating for and do loops. Now, consider the following app.

```
public class Loop {
  public static void main(String[] args) {
    for (int i = 0; i < 503; i++) {}
  }
}
```

This app terminates. However, if JPF is run with the above application properties file, then it produces output very similar to the above output. This shows that the listener produces false positives, that is, it reports an issue that does not actually exist. Hence, whenever the listener reports a suspicious loop, we still have to check whether it a nonterminating loop.

The listener `EndlessLoopDetector` can be customized by means of one property. The property `idle.max_backjumps` provides a lower bound on the number of iterations for a loop to be suspicious. For example, if we run JPF for the above app with the application properties file

```
target=Loop
classpath=.
listener=gov.nasa.jpf.listener.EndlessLoopDetector
idle.max_backjumps=1000
```

then JPF does not report any suspicious loops. The default value of the property `idle.max_backjumps` is 500.

# Chapter 5

# Using a Search Strategy

A search strategy determines in which order the states are visited and the transitions are traversed. Consider, for example, the following app.

```
1  import java.util.Random;
2
3  public class StateSpace {
4    public static void main(String[] args) {
5      Random random = new Random();
6      System.out.print("0 ");
7      if (!random.nextBoolean()) {
8        System.out.print("1 ");
9        if (!random.nextBoolean()) {
10         System.out.print("3 ");
11       } else {
12         System.out.print("4 ");
13         if (!random.nextBoolean()) {
14           System.out.print("5 ");
15         } else {
16           System.out.print("6 ");
17         }
18       }
19     } else {
20       System.out.print("2 ");
21     }
22   }
23 }
```

The corresponding state space is depicted in Figure 5.1. Suppose that the system is in initial state 0. In this state, the system executes line 5–7. If `random.nextBoolean()` of line 7 returns false, then the system transitions to state 1. Otherwise, it transitions to state 2. In state 1, the system executes line 8–9. If `random.nextBoolean()` of line 9 returns false, then the system transitions to state 3. Otherwise, it transitions to state 4. In state 3, the system executes line 10. In state 4, the system executes line 12–13. If `random.nextBoolean()` of line 13 returns false, then the system transitions to state 5. Otherwise, it transitions to state 6. In state 5 and 6, the system executes line 14 and 16, respectively. As a consequence, 0, 1, 2, 3, 4, 5 and 6 are printed when the system is in state 0, 1, 2, 3, 4, 5 and 6, respectively.

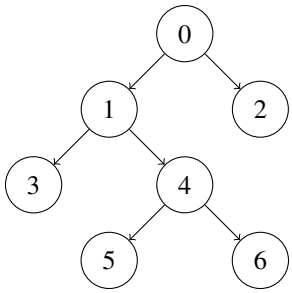



Figure 5.1: State-transition diagram.

## 5.1 Depth-First and Breadth-First Search

The output produced by JPF for the above app provides us some insight in the order in which the states are visited.

```
JavaPathfinder core system v8.0 (rev ...) - (C) 2005-2014 United States Government. All rig


====================================================== system under test
StateSpace.main()

====================================================== search started: 2/6/17 9:18 PM
0 1 3 4 5 6 2

====================================================== results
no errors detected
...
```

From the above we can conclude that JPF traverses the state space depth first. If we run JPF with the `-show` command line argument, we notice that the JPF property `search.class` is set as follows.

```
---------- Config contents
...
search.class = gov.nasa.jpf.search.DFSearch
...
```

The class `gov.nasa.jpf.search.DFSearch` implements depth-first search. This is the default search strategy. JPF also provides other search strategies. For example, the class `BFSHeuristic`, which is part of the package `gov.nasa.jpf.search.heuristic`, implements breadth-first search. If we set the JPF property `search.class` to `gov.nasa.jpf.search.heuristic.BFSHeuristic`, then JPF produces the following output.

```
JavaPathfinder core system v8.0 (rev ...) - (C) 2005-2014 United States Government. All rig


====================================================== system under test
StateSpace.main()

====================================================== search started: 2/6/17 9:18 PM
0 1 2 3 4 5 6

====================================================== results
```

```
no errors detected
...
```

The packages `gov.nasa.jpf.search` and `gov.nasa.jpf.search.heuristic` contain a few other search strategies.

## 5.2 Search Properties

There are a number of JPF properties that can be set for every search. With the JPF property `search.depth_limit` we can limit the depth of the search. This property can be assigned any integer value. The default value is `Integer.MAX_VALUE`. For example, if we set this property to 2 and use depth-first search, then JPF produces the following output.

```
JavaPathfinder core system v8.0 (rev ...) - (C) 2005-2014 United States Government. All rig


====================================================== system under test
StateSpace.main()

====================================================== search started: 2/6/17 9:18 PM
0 1
====================================================== search constraint
depth limit reached: 2

====================================================== snapshot
thread java.lang.Thread:{id:0,name:main,status:RUNNING,priority:5,isDaemon:false,lockCount
  call stack:
       at java.util.Random.nextBoolean(Random.java)
       at StateSpace.main(StateSpace.java:9)

2
====================================================== search constraint
depth limit reached: 2

====================================================== snapshot
no live threads

====================================================== results
no errors detected
...
```

The JPF property `search.min_free` captures the minimal amount of memory, in bytes, that needs to remain free. The default value is `1024 << 10`. By leaving some memory free, JPF can report that it ran out of memory and provide some useful statistics instead of simply throwing an `OutOfMemoryError`. For example, if we set this property to a large value and use depth-first search, then JPF produces the following output.

```
JavaPathfinder core system v8.0 (rev ...) - (C) 2005-2014 United States Government. All rig


====================================================== system under test
StateSpace.main()

====================================================== search started: 2/6/17 9:18 PM
0
```

```
====================================================== search constraint
memory limit reached
...
```

The JPF property `search.multiple_errors` tells us whether the search should report multiple errors (or just the first one). The default value is false, that is, by default only the first error is reported after which the search ends. Consider the following variation on the `StateSpace` app.

```
1  import java.util.Random;
2
3  public class StateSpace {
4    public static void main(String[] args) {
5      Random random = new Random();
6      System.out.print("0 ");
7      if (!random.nextBoolean()) {
8        System.out.print("1 ");
9        if (!random.nextBoolean()) {
10         System.out.print("3 ");
11         System.out.println(1 / 0);
12       } else {
13         System.out.print("4 ");
14         System.out.println(1 / 0);
15         if (!random.nextBoolean()) {
16           System.out.print("5 ");
17         } else {
18           System.out.print("6 ");
19         }
20       }
21     } else {
22       System.out.print("2 ");
23     }
24   }
25 }
```

This app differs from the one presented earlier in this chapter in line 11 and 14. In those lines, an exception is thrown. Since this exception is not caught in the code and JPF by default checks for uncaught exceptions, JPF reports errors for the above app. If the JPF property `search.multiple_errors` is not set in the application properties file, it has the default value false and, hence, only the first error is reported. In this case, JPF produces the following output.

```
JavaPathfinder core system v8.0 (rev ...) - (C) 2005-2014 United States Government. All rig


====================================================== system under test
StateSpace.main()

====================================================== search started: 11/28/18 5:05 PM
0 1 3
====================================================== error 1
gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
java.lang.RuntimeException
      at StateSpace.main(StateSpace.java:11)
```

```
====================================================== snapshot #1
thread java.lang.Thread:{id:0,name:main,status:RUNNING,priority:5,isDaemon:false,lockCount
  call stack:
       at StateSpace.main(StateSpace.java:11)


====================================================== results
error #1: gov.nasa.jpf.vm.NoUncaughtExceptionsProperty "java.lang.RuntimeException
at StateSpace.main(Sta..."
...
```

If the JPF property `search.multiple_errors` is set to true in the application properties file, JPF reports both errors, as is illustrated by the following output.

```
JavaPathfinder core system v8.0 (rev ...) - (C) 2005-2014 United States Government. All rig


====================================================== system under test
StateSpace.main()

====================================================== search started: 12/2/18 9:42 PM
0 1 3
====================================================== error 1
gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
java.lang.ArithmeticException: division by zero
at StateSpace.main(StateSpace.java:11)


====================================================== snapshot #1
thread java.lang.Thread:{id:0,name:main,status:RUNNING,priority:5,isDaemon:false,lockCount
call stack:
at StateSpace.main(StateSpace.java:11)

4
====================================================== error 2
gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
java.lang.ArithmeticException: division by zero
at StateSpace.main(StateSpace.java:14)


====================================================== snapshot #2
thread java.lang.Thread:{id:0,name:main,status:RUNNING,priority:5,isDaemon:false,lockCount
call stack:
at StateSpace.main(StateSpace.java:14)

2
====================================================== results
error #1: gov.nasa.jpf.vm.NoUncaughtExceptionsProperty "java.lang.ArithmeticException: divi
a..."
error #2: gov.nasa.jpf.vm.NoUncaughtExceptionsProperty "java.lang.ArithmeticException: divi
a..."
...
```

The JPF property `search.properties` specifies which properties are checked during the search. By default, this property has the values `gov.nasa.jpf.vm.NotDeadlockedProperty` and `gov.nasa.jpf.vm.NoUncaughtExceptionsProperty`. The former checks that an application has no deadlocks. We will come back to this in Chapter **??**. The latter checks that an application does not throw an exception that is not caught. A violation of this property indicates that an application throws an exception that is not caught. We have already seen several examples of that phenomenon in this and earlier chapters.

The class `NoOutOfMemoryErrorProperty`, which is part of the package `gov.nasa.jpf.vm`, checks that no `OutOfMemoryError` is thrown. We can *add* this property to `search.properties` by including in the application properties file the following.

```
search.properties+=,gov.nasa.jpf.vm.NoOutOfMemoryErrorProperty
```

# Chapter 6

# Using a Reporter

Like most components of JPF, its reporting system can be configured by setting the appropriate JPF properties. First of all, one has to set the `report.publisher` property. By default, its value is `console`. In that case, JPF produces for the `HelloWorld` application the output as presented in Chapter 2. The other currently supported value is `xml`. Consider the following application properties file.

```
target=HelloWorld
classpath=.
report.publisher=xml
```

JPF produces the output

```
Hello World!
```

and creates a file named `report.xml` with the content

```
<?xml version="1.0" ?>
<jpfreport>
</jpfreport>
```

In Chapter **??** we will show how new options for the `report.publisher` property can be added to JPF. Some listeners modify the output produced by JPF's reporting system. We will discuss an example in Section **??**. How to extend the reporting system within a listener is also covered in Chapter **??**.

## 6.1 Console

The output produced by the `console` option can be further customized by JPF properties. These properties specify what type of output is produced whenever an event is triggered by the search. In the table below, we provide the JPF properties and the corresponding events.

| event | property |
|---|---|
| `searchStarted` | `report.console.start` |
| `stateAdvanced` | `report.console.transition` |
| `searchProbed` | `report.console.probe` |
| `propertyViolated` | `report.console.property_violation` |
| `searchConstraintHit` | `report.console.constraint` |
| `searchFinished` | `report.console.finished` |

If we set all the above properties to empty values, then this results in the following application properties file.

```
target=HelloWorld
classpath=.
report.publisher=console
```

```
report.console.start=
report.console.transition=
report.console.probe=
report.console.property_violation=
report.console.constraint=
report.console.finished=
```

When we run JPF, it produces the output

```
Hello World!
```

To each of the last six properties, we can assign several values. We will discuss these next.

### 6.1.1 The `report.console.start` Property

To the property `report.console.start` we can assign any combination of the following values: `config`, `dtg`, `jpf`, `platform`, `sut`, and `user`. Let us first look at each separately. All other report properties, apart from `report.publisher`, are set to empty (which different from their default value).

#### The `config` Value

If the property `report.console.start` is set to `config`, then JPF produces the following output.

```
============================================================ JPF configuration
property source: /cs/home/franck/.jpf/site.properties
property source: /cs/home.franck/jpf/jpf-core/jpf.properties
property source: /cs/home/franck/examples/HelloWorld.jpf
properties:
branch_start=1
cg.boolean.false_first=true
...

============================================================ search started: 12/7/18 7:31 PM
Hello World!
```

Note that it produces output similar to that produced by the `-log` and `-show` command line arguments. It provides an overview of how JPF has been configured.

#### The `dtg` Value

If the property `report.console.start` is set to `dtg`, then JPF produces the following output.

```
started: Fri Dec 07 19:54:16 EST 2018

============================================================ search started: 12/7/18 7:54 PM
Hello World!
```

DTG stands for date-time group and captures the current time and date in a particular format.

#### The `jpf` Value

If the property `report.console.start` is set to `jpf`, then JPF produces the following output.

```
JavaPathfinder core system v8.0 (rev ...) - (C) 2005-2014 United States Government. All rig


============================================================ search started: 12/8/18 7:20 AM
Hello World!
```

Note that this is the banner we have seen at the start of every report.

#### The `platform` Value

If the property `report.console.start` is set to `platform`, then JPF produces output similar to the following.

```
============================================================ platform
hostname: indigo
arch: amd64/4
os: Linux/3.10.0-862.14.4.el7.x86_64
java: Oracle Corporation/1.8.0_181

============================================================ search started: 12/8/18 7:21 AM
Hello World!
```

#### The `sut` Value

If the property `report.console.start` is set to `sut`, short for system under test, then JPF produces the following output.

```
============================================================ system under test
HelloWorld.main()

============================================================ search started: 12/8/18 7:21 AM
Hello World!
```

#### The `user` Value

If the property `report.console.start` is set to `user`, then JPF produces output similar to the following.

```
user: franck

============================================================ search started: 12/8/18 7:34 AM
Hello World!
```

#### The Default Value

By default, the property `report.console.start` is set to `jpf,sut`. In that case, JPF produces the following output.

```
JavaPathfinder core system v8.0 (rev ...) - (C) 2005-2014 United States Government. All ri


============================================================ system under test
HelloWorld.main()

============================================================ search started: 12/8/18 7:48 AM
Hello World!
```

Note that the order in which the values are provided matters. For example, if we set the property `report.console.start` is set to `sut,jpf`, then JPF produces the following output.

```
============================================================ system under test
HelloWorld.main()
```

```
JavaPathfinder core system v8.0 (rev ...) - (C) 2005-2014 United States Government. All rig

====================================================== search started: 12/8/18 7:49 AM
Hello World!
```

### 6.1.2 The `report.console.transition` Property

To the property `report.console.transition` we can only assign the value `statistics` if the property `report.statistics` is set to true. Again, we assume that the other report properties, apart from `report.publisher`, are set to empty. In that case, JPF produces the following output.

```
Hello World!

====================================================== statistics
elapsed time:  00:00:00
states:        new=1,visited=0,backtracked=0,end=1
search:        maxDepth=1,constraints=0
choice generators: thread=1 (signal=0,lock=1,sharedRef=0,threadApi=0,reschedule=0), data=0
heap:          new=352,released=11,maxLive=0,gcCycles=1
instructions:  3175
max memory:    240MB
loaded code:   classes=61,methods=1329
```

By default, the empty value is assigned to the `report.console.transition` property.

### 6.1.3 The `report.console.probe` Property

To the property `report.console.probe` we can only assign the value `statistics`. That is also the default setting. If the property `report.probe_interval` is set to 1, that is, the search is probed every second, then JPF applied to the `Constraints` application of the beginning of Chapter 4 produces the following output. Again, we assume that the other report properties, apart from `report.publisher`, are set to empty.

```
====================================================== statistics
elapsed time:  00:00:01
states:        new=802,visited=798,backtracked=800,end=800
search:        maxDepth=800,constraints=0
choice generators: thread=1 (signal=0,lock=1,sharedRef=0,threadApi=0,reschedule=0), data=80
heap:          new=1195,released=329212,maxLive=1182,gcCycles=1600
instructions:  63078
max memory:    240MB
loaded code:   classes=70,methods=1657

====================================================== statistics
elapsed time:  00:00:02
states:        new=2403,visited=2398,backtracked=2400,end=2400
search:        maxDepth=2401,constraints=0
choice generators: thread=1 (signal=0,lock=1,sharedRef=0,threadApi=0,reschedule=0), data=24
heap:          new=2799,released=2907615,maxLive=2782,gcCycles=4801
instructions:  181727
max memory:    303MB
loaded code:   classes=70,methods=1657
...
```

That is, JPF outputs the statistics of the search every second.

### 6.1.4 The `report.console.property_violation` Property

To the property `report.console.property_violation` we can assign any combination of the following values: `error`, `output`, `snapshot`, `statistics`, and `trace`. By default, this property is set to `error,snapshot`.

To demonstrate this property we use the `UncaughtException` application of Chapter 2. We look at each value separately. Again, we assume that the other report properties, apart from `report.publisher`, are set to empty.

#### The `error` Value

If the property `report.console.property_violation` is set to `error`, then JPF produces the following output.

```
====================================================== error 1
gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
java.lang.ArithmeticException: division by zero
  at UncaughtException.main(UncaughtException.java:3)
```

JPF outputs the type of property that has been violated and some of its details.

#### The `output` Value

If the property `report.console.property_violation` is set to `output`, then JPF indicates that the `UncaughtException` application does not produce any output by producing the following output.

```
====================================================== output #1
no output
```

#### The `snapshot` Value

If the property `report.console.property_violation` is set to `snapshot`, then JPF produces the following output.

```
====================================================== snapshot #1
thread java.lang.Thread:{id:0,name:main,status:RUNNING,priority:5,isDaemon:false,lockCount
  call stack:
    at UncaughtException.main(UncaughtException.java:3)
```

It describes the main thread and its call stack.

#### The `statistics` Value

To the property `report.console.property_violation` we can only assign the value `statistics` if the property `report.statistics` is set to true. In that case, JPF produces the following output.

```
====================================================== statistics
elapsed time:  00:00:00
states:        new=1,visited=0,backtracked=0,end=0
search:        maxDepth=1,constraints=0
choice generators: thread=1 (signal=0,lock=1,sharedRef=0,threadApi=0,reschedule=0), data=0
heap:          new=366,released=0,maxLive=0,gcCycles=0
instructions:  3171
max memory:    240MB
loaded code:   classes=65,methods=1359
```

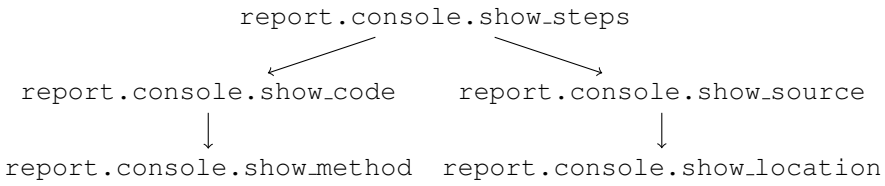



Figure 6.1: Dependencies between properties.

### The `trace` Value

Finally, we consider the value `trace` for the property `report.console.property_violation`. In this case, the output can be further refined by the following boolean properties:

- `report.console.show_cg`,
- `report.console.show_code`,
- `report.console.show_location`,
- `report.console.show_method`,
- `report.console.show_source`, and
- `report.console.show_steps`.

There are some dependencies between these six properties, which are depicted in Figure 6.1. An arrow from one property to another indicates that the value of the latter property is only relevant if the value of the former property is true. For example, the value of the property `report.console.show_location` is only relevant if the properties `report.console.show_steps` and `report.console.show_source` are both set to true. To illustrate these properties, we consider the `StateSpace` application of Section 5.2. We set the `sourcepath` property to the directory that contains the source code of the `StateSpace` class. We discuss some interesting combinations. As we will see in the examples below, the value `trace` ensures that a trace to the violation, that is, an abstraction of a partial execution ending in a state that violates the property of interest, is output. Again, we assume that the other report properties, apart from `report.publisher`, are set to empty.

### The `report.console.show_cg` Property

If we set the property `report.console.show_cg` to true and all the properties in Figure 6.1 to false, then JPF produces the following output.

```
0 1 3
====================================================== trace #1
------------------------------------------------------ transition #0 thread: 0
gov.nasa.jpf.vm.choice.ThreadChoiceFromSet {id:"ROOT" ,1/1,isCascaded:false}
------------------------------------------------------ transition #1 thread: 0
gov.nasa.jpf.vm.BooleanChoiceGenerator[[id="verifyGetBoolean",isCascaded:false,{>false,true
------------------------------------------------------ transition #2 thread: 0
gov.nasa.jpf.vm.BooleanChoiceGenerator[[id="verifyGetBoolean",isCascaded:false,{>false,true
```

For each transition, only the choice generator, which captures a nondeterministic choice, is represented. The first nondeterministic choice is related to concurrency and the other two are related to randomization.

### The `report.console.show_source` Property

If we set the properties `report.console.show_steps` and `report.console.show_source` to true and the other properties to false, then JPF produces the following output.

```
0 1 3
====================================================== trace #1
------------------------------------------------------ transition #0 thread: 0
   [3168 insn w/o sources]
  Random random = new Random();
   [2 insn w/o sources]
  Random random = new Random();
  System.out.print("0 ");
   [2 insn w/o sources]
  if (!random.nextBoolean()) {
   [1 insn w/o sources]
------------------------------------------------------ transition #1 thread: 0
   [2 insn w/o sources]
  if (!random.nextBoolean()) {
  System.out.print("1 ");
   [2 insn w/o sources]
  if (!random.nextBoolean()) {
   [1 insn w/o sources]
------------------------------------------------------ transition #2 thread: 0
   [2 insn w/o sources]
  if (!random.nextBoolean()) {
  System.out.print("3 ");
   [2 insn w/o sources]
  System.out.println(1 / 0);
```

The above trace consists of three transitions and for each transition the corresponding Java statements are output.

### The `report.console.show_location` Property

If we set the properties `report.console.show_steps`, `report.console.show_source`, and `report.console.show_location` to true and the other properties to false, then JPF produces the following output.

```
0 1 3
====================================================== trace #1
------------------------------------------------------ transition #0 thread: 0
   [3168 insn w/o sources]
  StateSpace.java:5      : Random random = new Random();
   [2 insn w/o sources]
  StateSpace.java:5      : Random random = new Random();
  StateSpace.java:6      : System.out.print("0 ");
   [2 insn w/o sources]
  StateSpace.java:7      : if (!random.nextBoolean()) {
   [1 insn w/o sources]
------------------------------------------------------ transition #1 thread: 0
   [2 insn w/o sources]
  StateSpace.java:7      : if (!random.nextBoolean()) {
  StateSpace.java:8      : System.out.print("1 ");
   [2 insn w/o sources]
```

```
  StateSpace.java:9      : if (!random.nextBoolean()) {
    [1 insn w/o sources]
------------------------------------------------------ transition #2 thread: 0
    [2 insn w/o sources]
  StateSpace.java:9      : if (!random.nextBoolean()) {
  StateSpace.java:10     : System.out.print("3 ");
    [2 insn w/o sources]
  StateSpace.java:11     : System.out.println(1 / 0);
```

The only difference with the previous configuration is that JPF outputs the line numbers as well.

### The `report.console.show_code` Property

If we set the properties `report.console.show_steps` and `report.console.show_code` to true and the other properties to false, then JPF produces the following output.

```
0 1 3
====================================================== trace #1
------------------------------------------------------ transition #0 thread: 0
  invokestatic java.lang.Boolean.<clinit>()V
  new java.lang.Boolean@bc
  dup
  ...
------------------------------------------------------ transition #1 thread: 0
  executenative JPF_java_util_Random.nextBoolean____Z
  nativereturn java.util.Random.nextBoolean()Z
  ifne 104
  ...
------------------------------------------------------ transition #2 thread: 0
  executenative JPF_java_util_Random.nextBoolean____Z
  nativereturn java.util.Random.nextBoolean()Z
  ifne 58
  ...
```

This configuration provides many more details of the trace. It abstracts at the level of bytecode instructions.

### The `report.console.show_method` Property

If we set the properties `report.console.show_steps`, `report.console.show_code`, and `report.console.show_method` to true and the other properties to false, then JPF produces the following output.

```
0 1 3
====================================================== trace #1
------------------------------------------------------ transition #0 thread: 0
  java.lang.Boolean.[<clinit>]
    invokestatic java.lang.Boolean.<clinit>()V
  java.lang.Boolean.<clinit>()V
    new java.lang.Boolean@bc
    dup
    ...
------------------------------------------------------ transition #1 thread: 0
  java.util.Random.nextBoolean()Z
    executenative JPF_java_util_Random.nextBoolean____Z
```

```
   nativereturn java.util.Random.nextBoolean()Z
  StateSpace.main([Ljava/lang/String;)V
   ifne 104
   ...
------------------------------------------------------ transition #2 thread: 0
  java.util.Random.nextBoolean()Z
   executenative JPF_java_util_Random.nextBoolean____Z
   nativereturn java.util.Random.nextBoolean()Z
  StateSpace.main([Ljava/lang/String;)V
   ifne 58
   ...
```

The difference with the previous configuration is that invocations to constructors and methods are also included.

### 6.1.5 The `report.console.constraint` Property

To the property `report.console.constraint` we can assign any combination of the following values: `constraint`, `output`, `snapshot`, `statistics`, and `trace`. By default, this property is set to `constraint,snapshot`. Again, we assume that the other report properties, apart from `report.publisher`, are set to empty.

To demonstrate this property we use the `StateSpace` application of Chapter 5. The property `search.depth_limit` is set to 2. As a result, the depth constraint is hit twice during the search.

#### The `constraint` Value

If the property `report.console.constraint` is set to `constraint`, then JPF produces the following output.

```
0 1
====================================================== search constraint
depth limit reached: 2
2
====================================================== search constraint
depth limit reached: 2
```

#### The `output` Value

If the property `report.console.constraint` is set to `output`, then JPF produces the following output.

```
0 1
====================================================== output
2 0 1
====================================================== output
```

Each time a constraint is hit, the output corresponding to the execution path leading to the constraint hit is output.

#### The `snapshot` Value

If the property `report.console.constraint` is set to `snapshot`, then JPF produces the following output.

```
0 1
====================================================== snapshot
thread java.lang.Thread:{id:0,name:main,status:RUNNING,priority:5,isDaemon:false,lockCount
  call stack:
    at java.util.Random.nextBoolean(Random.java)
    at StateSpace.main(StateSpace.java:9)
```

```
2
==================================================== snapshot
no live threads
```

Note that the second snapshot is taken at the end of the search when the main thread is not active any more.

### The `statistics` Value

To the property `report.console.constraint` we can only assign the value `statistics` if the property `report.statistics` is set to true. In that case, JPF produces the following output.

```
0 1
==================================================== statistics
elapsed time:  00:00:00
states:        new=2,visited=0,backtracked=0,end=0
search:        maxDepth=2,constraints=1
choice generators: thread=1 (signal=0,lock=1,sharedRef=0,threadApi=0,reschedule=0), data=1
heap:          new=358,released=2,maxLive=354,gcCycles=2
instructions:  3193
max memory:    240MB
loaded code:   classes=62,methods=1341
2
==================================================== statistics
elapsed time:  00:00:00
states:        new=3,visited=0,backtracked=1,end=1
search:        maxDepth=2,constraints=2
choice generators: thread=1 (signal=0,lock=1,sharedRef=0,threadApi=0,reschedule=0), data=1
heap:          new=360,released=12,maxLive=356,gcCycles=3
instructions:  3203
max memory:    240MB
loaded code:   classes=62,methods=1341
```

### The `trace` Value

Finally, we consider the value `trace` for the property `report.console.constraint`. Similar to the property `report.console.property_violation`, the output can be further refined by the following boolean properties: `report.console.show_cg`, `report.console.show_code`, `report.console.show_location`, `report.console.show_method`, `report.console.show_source`, and `report.console.show_steps`. Let us consider just one combination. If we set the properties `report.console.show_steps`, `report.console.show_source`, and `report.console.show_location` to true and the property `report.console.show_code` to false, then JPF produces the following output.

```
0 1
======================================================== trace
------------------------------------------------------ transition #0 thread: 0
   [3168 insn w/o sources]
  StateSpace.java:5      : Random random = new Random();
   [2 insn w/o sources]
  StateSpace.java:5      : Random random = new Random();
  StateSpace.java:6      : System.out.print("0 ");
   [2 insn w/o sources]
  StateSpace.java:7      : if (!random.nextBoolean()) {
   [1 insn w/o sources]
```

```
------------------------------------------------------ transition #1 thread: 0
   [2 insn w/o sources]
  StateSpace.java:7      : if (!random.nextBoolean()) {
  StateSpace.java:8      : System.out.print("1 ");
   [2 insn w/o sources]
  StateSpace.java:9      : if (!random.nextBoolean()) {
   [1 insn w/o sources]
2
====================================================== trace
------------------------------------------------------ transition #0 thread: 0
   [3168 insn w/o sources]
  StateSpace.java:5      : Random random = new Random();
   [2 insn w/o sources]
  StateSpace.java:5      : Random random = new Random();
  StateSpace.java:6      : System.out.print("0 ");
   [2 insn w/o sources]
  StateSpace.java:7      : if (!random.nextBoolean()) {
   [1 insn w/o sources]
------------------------------------------------------ transition #1 thread: 0
   [2 insn w/o sources]
  StateSpace.java:7      : if (!random.nextBoolean()) {
  StateSpace.java:22     : System.out.print("2 ");
   [2 insn w/o sources]
  StateSpace.java:24     : }
  StateSpace.java:3      : public class StateSpace {
```

### 6.1.6 The `report.console.finished` Property

To the property `report.console.finished` we can assign any combination of the following values: `statistics` and `result`. If the property `report.console.finished` is set to `statistics`, then JPF produces the following output.

```
Hello World!

====================================================== statistics
elapsed time:  00:00:00
states:        new=1,visited=0,backtracked=1,end=1
search:        maxDepth=1,constraints=0
choice generators: thread=1 (signal=0,lock=1,sharedRef=0,threadApi=0,reschedule=0), data=0
heap:          new=352,released=11,maxLive=0,gcCycles=1
instructions:  3175
max memory:    240MB
loaded code:   classes=61,methods=1329

====================================================== search finished: 12/8/18 8:15 AM
```

If the property `report.console.finished` is set to `result`, then JPF produces the following output.

```
Hello World!

====================================================== results
no errors detected
```

```
========================================================== search finished: 12/8/18 8:15 AM
```

By default, the property `report.console.finished` is set to `result,statistics`.

# Chapter 7

# Implementing a Listener

As we already mentioned in Chapter 4, JPF relies on event driven programming. In JPF events signal, for example, that a bytecode instruction has been executed, that a class has been loaded, or that the garbage collector has started. To handle these events, we implement listeners.

Both the search and JPF's virtual machine notify listeners of particular events. The methods corresponding to those events can be found in the interfaces `SearchListener` and `VMListener`, which are part of the packages `gov.nasa.jpf.search` and `gov.nasa.jpf.vm`, respectively. A listener implements either one of these or both interfaces.

Since usually we are only interested in a few events, we can provide the methods corresponding to the remaining events with a default implementation (since all methods are void, we can just provide a method with an empty body). To avoid coding these methods, JPF has already provided the class `SearchListenerAdapter` and the abstract class `ListenerAdapter`, which are part of the packages `gov.nasa.jpf.search` and `gov.nasa.jpf`, respectively. The former provides default implementations for all the methods specified in the `SearchListener` interface, and the latter provides default implementations for all the methods specified in the `SearchListener` and `VMListener` interface.

The interface `JPFListener`, which is part of package `gov.nasa.jpf`, represents listeners of JPF. Both `SearchListener` and `VMListener` extend this interface. The class `JPF` is the core of JPF. Among many other objects, it creates the listeners. The relationships among the above mentioned interfaces and classes is captured in the UML diagram depicted in Figure 7.1.

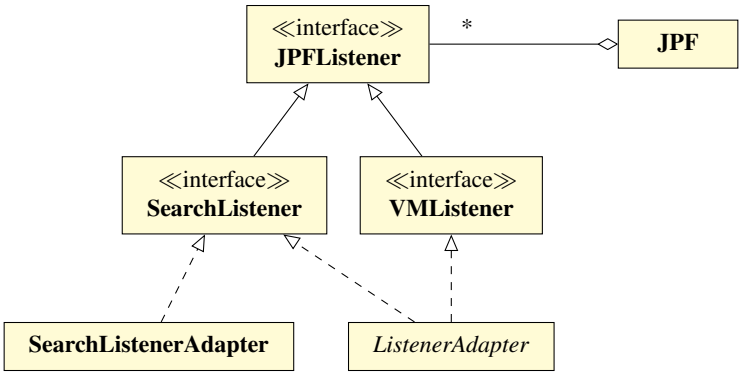



Figure 7.1: UML diagram of the listener interfaces and related classes.

## 7.1 The `SearchListener` Interface

The `SearchListener` interface contains methods that are related to events triggered by JPF's search. For example, when JPF starts its search, the method `searchStarted` is invoked. The `SearchListener` contains the following methods.

```
public interface SearchListener extends JPFListener {
  void stateAdvanced(Search search);
  void stateProcessed(Search search);
  void stateBacktracked(Search search);
  void statePurged(Search search);
  void stateStored(Search search);
  void stateRestored(Search search);

  void searchProbed(Search search);

  void propertyViolated(Search search);

  void searchConstraintHit(Search search);

  void searchStarted(Search search);
  void searchFinished(Search search);
}
```

The first six methods are used by the search to signal something about the current state of the search. The method `stateAdvanced` signals that the search just transitioned to the current state by executing the sequence of bytecode instructions associated with the transition. The method `stateProcessed` signals that the search has fully explored the current state. The method `stateBacktracked` signals that the search just backtracked to the current state. The method `statePurged` signals that the current state has been purged from the search, so that it will not be visited in the remainder of the search.[1] The method `stateStored` signals that the current state has been stored, as happens, for example, in breadth-first search. The method `stateRestored` signals that the current state, which was stored earlier in the search, has just been restored.

The method `searchProbed` signals that there was a probe request (see Section 6.1.3). The method `propertyViolated` signals that a violation of some property was detected while transitioning to the current state. The method `searchConstraintHit` signals that a constraint has been reached. Constraints such as the depth of the search and the amount of memory used by JPF are usually specified in a properties file. The methods `searchStarted` and `searchFinished` signal that the search has started and finished, respectively.

Note that these methods receive a reference to a `Search` object as an argument. This `Search` object contains information about the search of the state space by JPF. As we already mentioned, JPF invokes the method `searchStarted` at the start of the search with a reference to a `Search` object. When implementing the `searchStarted` in a listener, this `Search` object can be used to get information about the search of JPF.

## 7.2 Printing the Search Events

We start with an example listener that simply prints the sequence of events that have been triggered during JPF's search. For each event, it prints the sort of event and the ID of the state of the search in which the event occurs. We name our class `SearchEvents`. It implements the interface `SearchListener`. Therefore, we have the following class header.

```
public class SearchEvents implements SearchListener
```

[1] We are unaware of any search that purges states.

We implement each method of the `SearchListener` interface. For example, the `stateAdvanced` method is implemented as follows.

```
public void stateAdvanced(Search search) {
  System.out.println("advanced to state " + search.getStateId());
}
```

Recall that the `Search` object can be used to get information about the search. In particular, its `getStateId` returns the ID of the current state of the search.

For example, consider the following application (the comments in the code will become clear later).

```
import java.util.Random;

public class StateSpace {
  // state -1
  public static void main(String[] args) {
    // state 0
    Random random = new Random();
    if (!random.nextBoolean()) {
      // state 1
    } else {
      // state 2
      if (!random.nextBoolean()) {
        // state 1
      } else {
        // state 1
      }
    }
  }
}
```

If we run JPF on the above application, then the `SearchEvents` listener produces the following output.[2]

```
1  search started in state -1
2  advanced to state 0
3  advanced to state 1
4  state 1 processed
5  backtracked to state 0
6  advanced to state 2
7  advanced to state 1
8  backtracked to state 2
9  advanced to state 1
10 backtracked to state 2
11 state 2 processed
12 backtracked to state 0
13 state 0 processed
14 backtracked to state -1
15 state -1 processed
16 search finished in state -1
```

The state-transition diagram is depicted in Figure 8.1. The search starts in state $-1$ (line 1). Subsequently, it advances to state 0 (line 2) and then to state 1 (line 3). State 1 is a final state. Since state 1 has no outgoing transitions (that have not been traversed yet), the search signals that the state has been processed (line 4). Next, it backtracks to

[2]The `DFSearch` discussed in Chapter 5 does not produce all of the events (line 4 is missing), but the `DFSearch` that we will develop in Chapter 9 does.

state 0 (line 5). From state 0, the search advances along the still unexplored transition to state 2 (line 6). From state 2 it advances to state 1 (line 7). This state has already been fully explored so the search backtracks to state 2 (line 8). Since state 2 has another, still unexplored, transition to state 1, the search advances along this transition to state 1 (line 9). Next, the search backtracks to state 2 (line 10), after which it signals that this state has been processed (line 11), and similarly for state 0 (line 12–13) and state $-1$ (line 14–15). At that point, the search terminates (line 16).

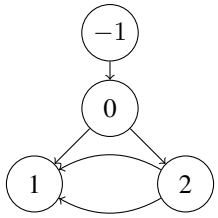



Figure 7.2: state-transition diagram of `StateSpace`.

State 0 is often considered the initial state. The bytecode instructions associated with the transition from state $-1$ to state 0 correspond to initializing the virtual machine and invoking the `main` method.

If we use breadth-first search for the above application, then the `SearchEvents` listener produces the following output.[3]

```
1  search started in state -1
2  state -1 stored
3  state -1 restored
4  advanced to state 0
5  state 0 stored
6  backtracked to state -1
7  state -1 processed
8  state 0 restored
9  advanced to state 1
10 state 1 processed
11 backtracked to state 0
12 advanced to state 2
13 state 2 stored
14 backtracked to state 0
15 state 0 processed
16 state 2 restored
17 advanced to state 1
18 backtracked to state 2
19 advanced to state 1
20 backtracked to state 2
21 state 2 processed
22 search finished in state 2
```

The main difference with depth-first search is that breadth-first search stores and restores states.

## 7.3 Printing the State Space

As a first example, we implement a listener which prints the states and transitions visited by the search in the following simple format:

[3]The `BFSHeuristic` discussed in Chapter 5 does not produce all of the events in this order (line 1 and 2 are reversed, and line 3 and 10 are missing), but the `BFSearch` that we will develop in Chapter 9 does.

```
0 -> 1
1 -> 2
0 -> 3
3 -> 4
4 -> 2
```

The above tells us that the search started in the initial state 0 and made a transition to a new state 1. From state 1 it made a transition to a new state 2. Next, it made a transition from state 0 to state 2. Et cetera. We name our class `StateSpacePrinter`. In our `StateSpacePrinter`, we are only interested in the events signalling a change of state. Therefore, we implement the `SearchListener` interface. Since we are not interested in all event notifications by the search, we start from default implementations by extending the `SearchListenerAdapter` class. Hence, we arrive at the following method header.

```
public class StateSpacePrinter extends SearchListenerAdapter
  implements SearchListener
```

The only three methods that signal a state change are `stateAdvanced`, `stateBacktracked` and `stateRestored`. In order to print a transition, we need both the source and target state of the transition. We introduce attributes to keep track of the source and target state. In JPF each state has a unique identifier, which is a nonnegative integer.[4]

```
private int source;
private int target;
```

We initialize these attributes in the constructor to -1.

```
public StateSpacePrinter() {
  this.source = -1;
  this.target = -1;
}
```

We have left to implement the methods `stateAdvanced`, `stateBacktracked` and `stateRestored`. These can be implemented in the following straightforward way.

```
public void stateAdvanced(Search search) {
  this.source = this.target;
  this.target = search.getStateId();
  if (this.source != -1) {
    System.out.printf("%d -> %d%n", source, target);
  }
}

public void stateBacktracked(Search search) {
  this.target = search.getStateId();
}

public void stateRestored(Search search) {
  this.target = search.getStateId();
}
```

In our implementation of the methods we use the parameter `search`. In particular, we use `search.getStateId()` to get the identifier of the current state of the search.

[4] There is a state with identifier $-1$ and a transition from this state to the initial state 0. This transition captures the bytecode instructions that are executed by the virtual machine until it reaches the beginning of the `main` method.

## 7.4 The State Space in DOT Format

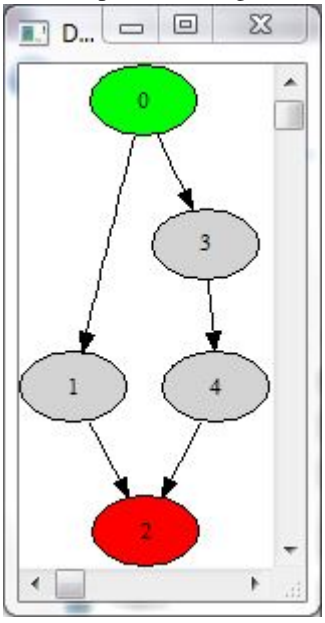
Rather than printing the transitions in the terminal and, hence, interleaving them with the output produced by the system under test, we can write the transitions to a file. Also, instead of representing them as text, we can save the states and transitions in DOT format. This allows us to use tools such as Graphviz, as discussed in Section 4.1, to layout and view the states and transitions. We will develop such a listener next. We colour the initial state green and all final states red. The textual representation

```
0 -> 1
1 -> 2
0 -> 3
3 -> 4
4 -> 2
```

where 0 is the initial state and 2 is a final state corresponds to the following DOT representation.

```
digraph statespace {
node [style=filled]
0 [fillcolor=green]
0 -> 1
1 -> 2
2 [fillcolor=red]
0 -> 3
3 -> 4
4 -> 2
2 [fillcolor=red]
}
```

This DOT representation gives rise to the graphical representation of the state space depicted below.




To write to a file, we introduce an attribute of type `PrintWriter` as follows.

```
private PrintWriter writer;
```

We initialize this attribute in the `searchStarted` method. As the name of the DOT file, we use the name of the system under test with ".dot" as suffix. The name of the system under test can be obtained using the method invocation

`search.getVM().getSUTName()`. Whenever the creation of the `PrintWriter` fails, we want to print an error message and subsequently terminate JPF. The latter is done using `search.terminate()`.

When the search starts we also write

```
digraph statespace {
node [style=filled]
0 [fillcolor=green]
```

to the file. In JPF, 0 is the identifier of the initial state. This is accomplished as follows.

```
public void searchStarted(Search search) {
  String name = search.getVM().getSUTName() + ".dot";
  try {
    this.writer = new PrintWriter(name);
    this.writer.println("digraph statespace {");
    this.writer.println("node [style=filled]");
    this.writer.println("0 [fillcolor=green]");
  } catch (FileNotFoundException e) {
    System.out.println("Listener could not write to file " + name);
    search.terminate();
  }
}
```

The method `stateAdvanced` is modified as follows.

```
public void stateAdvanced(Search search) {
  this.source = this.target;
  this.target = search.getStateId();
  if (this.source != -1) {
    this.writer.printf("%d -> %d%n", source, target);
  }
  if (search.isEndState()) {
    writer.printf("%d [fillcolor=red]%n", target);
  }
}
```

When the search terminates, we write } to the file and close the `PrintWriter` object and release any system resources associated with it. This is done by adding the following method.

```
public void searchFinished(Search search) {
  this.writer.println("}");
  this.writer.close();
}
```

## 7.5 The VMListener Interface

The `VMListener` interface contains methods that are related to events triggered by JPF's virtual machine. For example, whenever JPF executes a bytecode instruction, the method `executeInstruction` is invoked. The VMListener contains the following methods.

```
public interface VMListener extends JPFListener {
  void vmInitialized(VM vm);

  void executeInstruction(VM vm, ThreadInfo currentThread,
    Instruction instructionToExecute);
```

```
  void instructionExecuted(VM vm, ThreadInfo currentThread,
    Instruction nextInstruction, Instruction executedInstruction);

  void threadStarted(VM vm, ThreadInfo startedThread);
  void threadBlocked(VM vm, ThreadInfo blockedThread, ElementInfo lock);
  void threadWaiting(VM vm, ThreadInfo waitingThread);
  void threadNotified(VM vm, ThreadInfo notifiedThread);
  void threadInterrupted(VM vm, ThreadInfo interruptedThread);
  void threadTerminated(VM vm, ThreadInfo terminatedThread);
  void threadScheduled(VM vm, ThreadInfo scheduledThread);

  void loadClass(VM vm, ClassFile cf);
  void classLoaded(VM vm, ClassInfo loadedClass);

  void objectCreated(VM vm, ThreadInfo currentThread, ElementInfo newObject);
  void objectReleased(VM vm, ThreadInfo currentThread, ElementInfo releasedObject);
  void objectLocked(VM vm, ThreadInfo currentThread, ElementInfo lockedObject);
  void objectUnlocked(VM vm, ThreadInfo currentThread, ElementInfo unlockedObject);
  void objectWait(VM vm, ThreadInfo currentThread, ElementInfo waitingObject);
  void objectNotify(VM vm, ThreadInfo currentThread, ElementInfo notifyingObject);
  void objectNotifyAll(VM vm, ThreadInfo currentThread, ElementInfo notifyingObject);
  void objectExposed(VM vm, ThreadInfo currentThread, ElementInfo fieldOwnerObject,
    ElementInfo exposedObject);
  void objectShared(VM vm, ThreadInfo currentThread, ElementInfo sharedObject);

  void gcBegin(VM vm);
  void gcEnd(VM vm);

  void exceptionThrown(VM vm, ThreadInfo currentThread, ElementInfo thrownException);
  void exceptionBailout(VM vm, ThreadInfo currentThread);
  void exceptionHandled(VM vm, ThreadInfo currentThread);

  void choiceGeneratorRegistered(VM vm, ChoiceGenerator<?> nextCG,
    ThreadInfo currentThread, Instruction executedInstruction);
  void choiceGeneratorSet(VM vm, ChoiceGenerator<?> newCG);
  void choiceGeneratorAdvanced(VM vm, ChoiceGenerator<?> currentCG);
  void choiceGeneratorProcessed(VM vm, ChoiceGenerator<?> processedCG);

  void methodEntered(VM vm, ThreadInfo currentThread, MethodInfo enteredMethod);
  void methodExited(VM vm, ThreadInfo currentThread, MethodInfo exitedMethod);
}
```

Rather than discussing all the above methods, we present a number of listeners that use some of those methods.

## 7.6 Printing the Bytecode Mnemonics

As a simple example, we implement a listener that prints the mnemonics of the bytecode instructions executed by JPF. For example, if we apply this listener to the `HelloWorld` app, then JPF produces the following output.

```
JavaPathfinder core system v8.0 (rev ...) - (C) 2005-2014 United States Government. All rig
```

```
========================================================== system under test
HelloWorld.main()

========================================================== search started: 11/15/18 12:01 PM
invokeclinit
new
dup
iconst_1
invokespecial
aload_0
invokespecial
...
```

We name our listener `Mnemonics`. In this class, we are only interested in the events signalling the execution of a bytecode instruction. Therefore, we implement the VMListener interface. Since we are only interested in a single event, we start from default implementations by extending the `ListenerAdapter` class. Hence, we arrive at the following class header.

```
public class Mnemonics extends ListenerAdapter implements VMListener
```

The method `executeInstruction` signals which instruction JPF will execute next. In JPF, instructions are represented as instances of the class `Instruction`, which is part of the package `gov.nasa.jpf.vm`. The next instruction is provided as an argument of the `executeInstruction` method. The class `Instruction` contains the method `getMnemonic` that returns the mnemonic of the instruction. Hence, the `executeInstruction` method can be implemented as follows.

```
public void executeInstruction(VM vm, ThreadInfo currentThread, Instruction instruction) {
  System.out.println(instruction.getMnemonic());
}
```

## 7.7 Timing the Garbage Collector

Next, we develop a listener that times the garbage collector every time it is invoked by JPF. The methods `gcBegin` and `gcEnd` signal the beginning and end of garbage collecting. Both are part of the `VMListener` interface. Hence, we start with the following class header.

```
public class Garbage extends ListenerAdapter implements VMListener
```

To keep track of the current time (in milliseconds), we introduce the following attribute.

```
private long time;
```

Whenever the garbage collector begins, we save the current time in the attribute.

```
public void gcBegin(VM vm) {
  this.time = System.currentTimeMillis();
}
```

Whenever the garbage collector ends, we use the current time and the time when the garbage collector began, saved in the attribute, to compute the number of milliseconds the garbage collector has been active and print the result.

```
public void gcEnd(VM vm) {
  System.out.printf("Garbage collection: %d milliseconds%n",
    System.currentTimeMillis() - time);
}
```

## 7.8 Amount of Nondeterminism

In our next example, we determine the amount nondeterminism, that is, the maximum number of transitions that leave a state. Since this information has to be extracted from JPF's virtual machine, we implement the `VMListener` interface. Since we can only print this information at the end of the search, we also implement the `SearchListener` interface. Hence, we extend the `ListenerAdapter` class.

```
public class AmountOfNondeterminism extends ListenerAdapter
  implements SearchListener, VMListener
```

We use the attribute `max` to keep track of the maximal amount of nondeterminism observed so far.

```
private int max;
```

In the constructor, we initialize this attribute to zero.

```
public AmountOfNondeterminism() {
  this.max = 0;
}
```

Whenever JPF encounters a nondeterministic choice, its virtual machine sends a notification which amounts to an invocation of the `choiceGeneratorSet` method. This method receives two arguments: a reference to JPF's virtual machine, represented by the `VM` class, and the `ChoiceGenerator` object that represents the nondeterministic choice in JPF. The number of nondeterministic choices of this `ChoiceGenerator` is returned by the method `getTotalNumberOfChoices`. Hence, the method `choiceGeneratorSet` can be implemented as follows.

```
public void choiceGeneratorSet(VM vm, ChoiceGenerator<?> newCG) {
  this.max = Math.max(this.max, newCG.getTotalNumberOfChoices());
}
```

The end of the search is signalled by the `searchFinished` method of the `SearchListener` interface. To print the maximum amount of nondeterminism, we implement the `searchFinished` method as follows.

```
public void searchFinished(Search search) {
  System.out.printf("%n====================================================%n");
  System.out.printf("maximum amount of nondeterminism: %d choice(s)%n", this.max);
}
```

## 7.9 A Simple Profiler

Let us develop a listener that, at the end of the search, prints the (array) objects that have been created and the methods that have been executed by JPF. For example, if we apply this listener to the `HelloWorld` app, then JPF produces the following output.

```
JavaPathfinder core system v8.0 (rev ...) - (C) 2005-2014 United States Government. All ri


============================================================ system under test
HelloWorld.main()

============================================================ search started: 05/12/18 10:05 AM
Hello World!

Created arrays
120   char
2     java.lang.String
```

```
2     java.util.Hashtable$Entry
1     java.lang.Thread$State
...
Created other objects
116   java.lang.String
78    java.lang.Class
11    java.util.Hashtable$Entry
...
Invoked methods
24    <init> of java.lang.Object
12    hashCode of java.lang.String
11    put of java.util.Hashtable
...

====================================================== results
no errors detected
...
```

This tells us that 120 character arrays and 116 `String`s have been created and the constructor of the `Object` class has been executed 24 times.

In order to implement this listener, we need to keep track of several events: the creation of an object, the execution of a method, and the end of the search. These are signalled by the methods `objectCreated` and `methodEntered` of the interface `VMListener`, and the method `searchFinished` of the interface `SearchListener`. Since the class `ListenerAdapter` provides default implementations for the interfaces `VMListener` and `SearchListener`, we create the class `Profiler` with the following header.

```
public class Profiler extends ListenerAdapter implements
  SearchListener, VMListener
```

In JPF, objects are represented by instances of the class `ElementInfo` and methods are represented by instances of the class `MethodInfo`. The type of an object is represented in JPF by instances of the class `ClassInfo`. All these classes are part of the package `gov.nasa.jpf.vm`. In order to keep track of the objects that have been created and the methods that have been executed by JPF and the number of times that these objects have been created and these methods have been executed, we introduce the following attributes.

```
private Map<ClassInfo, Integer> arrays;
private Map<ClassInfo, Integer> objects;
private Map<MethodInfo, Integer> methods;
```

As we will see later, for array objects we use their base types, represented by `ClassInfo` objects, as keys in the map `arrays`. Initially, JPF has neither created any object nor executed any methods. Hence, we initialize the attributes `arrays`, `objects`, and `methods` in the constructor of our `Profiler` class as follows.

```
public Profiler() {
  this.arrays = new HashMap<ClassInfo, Integer>();
  this.objects = new HashMap<ClassInfo, Integer>();
  this.methods = new HashMap<MethodInfo, Integer>();
}
```

Whenever an object is created by JPF's virtual machine, the method `objectCreated` method is invoked. This method takes a `ElementInfo` object, representing the object that is created, as an argument. In our listener, we implement this method by incrementing the number of objects that have been created of its given type. We distinguish between objects that are arrays and that are not, using the method `isArray` of the class `ElementInfo`. To get the type of an object, that is, to get the `ClassInfo` object corresponding to an `ElementInfo` object, we use the method `getClassInfo` of the class `ElementInfo`. For a `ClassInfo` object that represents the type of an

array, its base type can be obtained by means of the `getComponentClassInfo` method of the `ClassInfo` class. Hence, whenever an object is created, the attributes `arrays` and `objects` can be updated as follows.

```
public void objectCreated(VM vm, ThreadInfo currentThread, ElementInfo newObject) {
  if (newObject.isArray()) {
    ClassInfo clazz = newObject.getClassInfo().getComponentClassInfo();
    if (this.arrays.containsKey(clazz)) {
      this.arrays.put(clazz, this.arrays.get(clazz) + 1);
    } else {
      this.arrays.put(clazz, 1);
    }
  } else {
    ClassInfo clazz = newObject.getClassInfo();
    if (this.objects.containsKey(clazz)) {
      this.objects.put(clazz, this.objects.get(clazz) + 1);
    } else {
      this.objects.put(clazz, 1);
    }
  }
}
```

Whenever a method is executed, the method `methodEntered` is triggered. This method takes a `MethodInfo` object, representing the method that is executed, as an argument. Hence, whenever a method is executed the attribute `methods` can be updated as follows.

```
public void methodEntered(VM vm, ThreadInfo thread, MethodInfo method) {
  if (this.methods.containsKey(method)) {
    this.methods.put(method, this.methods.get(method) + 1);
  } else {
    this.methods.put(method, 1);
  }
}
```

In order to print the objects and methods from highest to lowest frequency, we need to sort the entries of the maps by value. For that purpose, we pair their names and frequencies so that they can be sorted by frequency.

```
private class Pair implements Comparable<Pair> {
  private final int frequency;
  private final String name;

  public Pair(int frequency, String name) {
    this.frequency = frequency;
    this.name = name;
  }

  public String toString() {
    return this.frequency + "\t" + this.name;
  }

  public int compareTo(Pair other) {
    return other.frequency - this.frequency;
  }
}
```

The end of the search is signalled by the `searchFinished` method. At that point, we can print the desired output as follows.

```
public void searchFinished(Search search) {
  List<Pair> list = new ArrayList<Pair>();
  for (ClassInfo clazz : this.arrays.keySet()) {
    list.add(new Pair(this.arrays.get(clazz), clazz.getName()));
  }
  Collections.sort(list);
  System.out.println("Created arrays");
  for (Pair pair : list) {
    System.out.println(pair);
  }

  list = new ArrayList<Pair>();
  for (ClassInfo clazz : this.objects.keySet()) {
    list.add(new Pair(this.objects.get(clazz), clazz.getName()));
  }
  Collections.sort(list);
  System.out.println("Created other objects");
  for (Pair pair : list) {
    System.out.println(pair);
  }

  list = new ArrayList<Pair>();
  for (MethodInfo method : this.methods.keySet()) {
    list.add(new Pair(this.methods.get(method), method.getName() + " of " + method.getClassI
  }
  Collections.sort(list);
  System.out.println("Invoked methods");
  for (Pair pair : list) {
    System.out.println(pair);
  }
}
```

## 7.10 Parametrizing a Listener

Now assume that we want to parametrize the listener, that is, we want to customize it by means of some JPF property. Suppose we want to give the user the opportunity to specify which string is used to separate the states of a transition in our `StateSpacePrinter`. By default, this separator is "`->`". We introduce the JPF property `statespaceprinter.separator`. This property can be set by the user in a properties file or as a command line argument. For example, the user can add

```
statespaceprinter.separator=-->
```

to the application properties file. To our class, we add the attribute

```
private String separator;
```

We initialize this attribute in the constructor. Recall that a `Config` object contains the JPF properties. The `Config` class contains methods to extract the values of JPF properties. For example, the method `getString` extracts the string value of some JPF property. The method takes two arguments, the name of the JPF property and the default value (in case the `Config` object does not contain any value for the JPF property). For example,

```
config.getString("statespaceprinter.separator", "->");
```

returns the value of the JPF property `statespaceprinter.separator` if that property is defined in the `Config` object `config`, and returns "->" otherwise.

We change the signature of the constructor: we add a parameter of type `Config`. When JPF constructs a `StateSpacePrinter`, it provides a `Config` object as argument to the constructor. Hence, our modified constructor now looks as follows.

```
public StateSpacePrinter(Config config) {
  this.source = -1;
  this.separator = config.getString("statespaceprinter.separator", "->");
}
```

A constructor of a listener can either take no arguments, one argument of type `Config`, or two arguments, the first of type `Config` and the second of type `VM`. We have already seen examples of the first and second case above. We will present an example of the third case later.

Of course, we also have to modify the `stateAdvanced` method as follows.

```
public void stateAdvanced(Search search) {
  this.source = this.target;
  this.target = search.getStateId();
  if (this.source != -1) {
    System.out.printf("%d %s %d%n", this.source, this.separator, this.target);
  }
}
```

## 7.11 Compiling a new Listener

The `StateSpacePrinter` listener relies on several classes that are part of JPF such as, for example, `ListenerAdapter`, which is part of the package `gov.nasa.jpf`. The bytecode of these classes, which are needed to compile the `StateSpacePrinter` class, can be found in the jar file `jpf.jar`. This jar file can be found in the `build` directory of `jpf-core`. If `jpf-core` has been installed in the directory `/cs/home/franck/projects/jpf/jpf-core`, then the `StateSpacePrinter` class can be compiled as follows.

```
javac -cp /cs/home/franck/projects/jpf/jpf-core/build/jpf.jar;. StateSpacePrinter.java
```

In Eclipse and NetBeans, if the listener `StateSpacePrinter` is in a different project than `jpf-core`, then the jar file `jpf.jar` needs to be added to to the project.

## 7.12 Using a new Listener: `classpath` versus `native_classpath`

As we already mentioned earlier, JPF is implemented as a JVM. Therefore, JPF has a classpath. This classpath can be set in a configuration file using the `classpath` property. JPF's classpath should contain the Java bytecode of the classes that need to be model checked. That is, it should contain the bytecode of the system under test and any classes used by it.

Since JPF is implemented in Java, it runs on top of a JVM, which we will call the host JVM. This host JVM has a classpath as well. This classpath can be set in a configuration file using the `native_classpath` property. It should contain the bytecode of the classes that are needed to run JPF and its extensions, such as listeners.

Assume that the bytecode of the listener `StateSpacePrinter` can be found in the directory `/cs/home/franck/projects/`. Suppose that the bytecode of the app `HelloWorld` can be found in the directory `/cs/home/franck/projects/code/`. In that case, we can run JPF on the `HelloWorld` app with the `StateSpacePrinter` listener using the following application configuration file.

```
target=HelloWorld
classpath=/cs/home/franck/projects/code/
```

```
native_classpath=/cs/home/franck/projects/listeners/
listener=StateSpacePrinter
```

# Chapter 8

# Implementing a Reporter

## 8.1 The `PublisherExtension` Interface

Rather than using print statements as in the `StateSpacePrinter`, we can also use JPF's report system to provide the results of the verification effort to the user. To exploit the report system, we implement the interface `PublisherExtension`, which is part of the package `gov.nasa.jpf.report`.

```
public interface PublisherExtension {
  void publishStart(Publisher publisher);
  void publishTransition(Publisher publisher);
  void publishPropertyViolation(Publisher publisher);
  void publishConstraintHit(Publisher publisher);
  void publishFinished(Publisher publisher);
  void publishProbe(Publisher publisher);
}
```

The `PublisherExtension` interface contains methods which are invoked whenever a particular event occurs. For example, the `publishTransition` method is invoked whenever a transition is traversed by JPF. These methods receive an object of type `Publisher` as argument. The class `Publisher`, which is part of the package `gov.nasa.jpf.report`, contains methods that provide information about how the data is formatted by JPF. For example, the method `getOut` returns the `PrintWriter` object which is used by JPF to print the data. The type of `PrintWriter` object that JPF uses can be set by means of the JPF property `report.publisher`. Its default value is `console`, but JPF can also produce XML by setting

```
report.publisher=xml
```

in the application properties file.

## 8.2 The State Space in XML Format

The earlier mentioned class `ListenerAdapter` provides default implementations of all method specifications in the interface `PublisherExtension`. Hence, the header of our `StateSpacePrinter` class becomes as follows.

```
public class StateSpacePrinter extends ListenerAdapter
  implements SearchListener, PublisherExtension
```

Since we want to publish data whenever a transition is traversed, we implement the `publishTransition` method. This method can be implemented as follows.

```
public void publishTransition(Publisher publisher) {
  PrintWriter out = publisher.getOut();
```

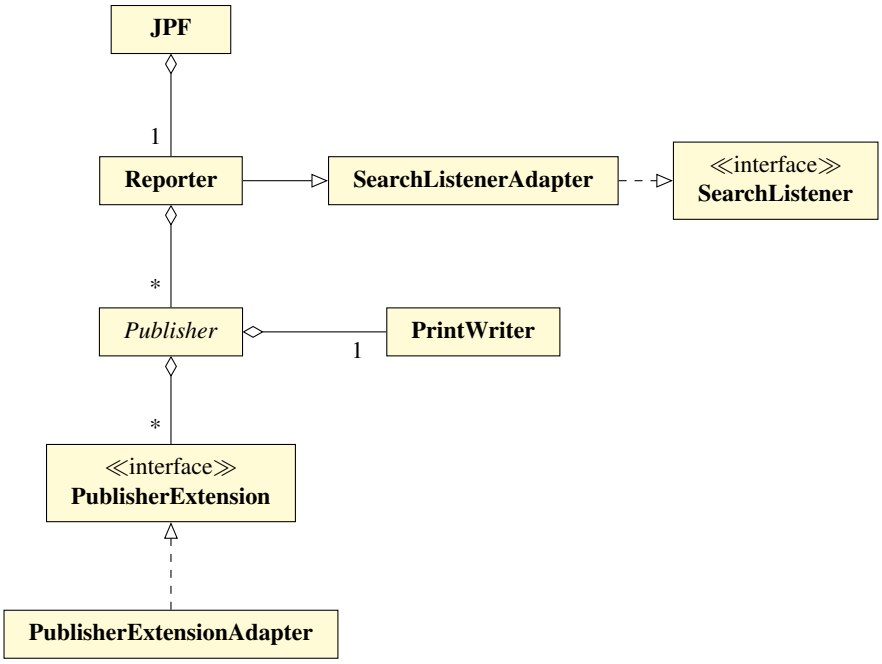



Figure 8.1: UML diagram of the listener interfaces and related classes.

```
  if (this.source != -1) {
    ???
  }
}
```

Now that we have moved the printing of the transition to the `publishTransition` method, we can remove it from the `stateAdvanced` method. The latter method can be simplified to the following.

```
public void stateAdvanced(Search search) {
  this.source = this.target;
  this.target = search.getStateId();
}
```

We have left to register our `StateSpacePrinter` class as a `PublisherExtension`. This is done in the constructor as follows.

```
public StateSpacePrinter(Config config, JPF jpf) {
  this.source = -1;
  this.target = -1;
  jpf.addPublisherExtension(Publisher.class, this);
}
```

Note that this constructor takes two arguments: a `Config` object and a `JPF` object. The latter is needed for JPF's reporting system.

# Chapter 9

# Implementing a Search Strategy

In this chapter we provide a recipe for implementing a search strategy within Java Pathfinder (JPF). As a running example, we use depth-first search (DFS). We name our class `DFSearch`.

## 9.1 The Structure of the Class

The `Search` class, which resides in the package `gov.nasa.jpf.search`, contains numerous attributes and methods that may be of use for implementing a search strategy. Hence, we extend this class.

```
import gov.nasa.jpf.search.Search;

public class DFSearch extends Search {
  ...
}
```

The constructor of the `Search` class takes two arguments. The first argument is a `Config` object. The class `Config` is part of the package `gov.nasa.jpf`. The `Config` object contains the JPF properties. These properties can be set, for example, in the application properties file and the `jpf.properties` file. The second argument is a `VM` object. The package `gov.nasa.jpf.vm` contains the `VM` class. The `VM` object provides the search a reference to JPF's virtual machine. To properly initialize the `Search` object, we add the following constructor to our `DFSearch` class (and import the classes `Config` and `VM`).

```
public DFSearch(Config config, VM vm) {
  super(config, vm);
}
```

If JPF is configured to use our `DFSearch`, JPF will construct a `DFSearch` object with a `Config` object capturing JPF's configuration and a `VM` object representing JPF's virtual machine. For `DFSearch` to properly interact with JPF, its constructor has to be declared public.

The `Search` class is abstract. It contains the abstract method `search`. This method drives the search. It visits the states of the system under test in a systematic way by traversing transitions. In our `DFSearch` we implement the `search` method. We develop our implementation of the `search` method in a number of steps.

## 9.2 The Basic Search

To visit the states, the `Search` class provides the following two methods. The `backtrack` method returns the search to the source state of the transition along which the current state was discovered by the search. The method returns a boolean: whether the `backtrack` was successful. A backtrack fails, for example, in the initial state. The `forward` method moves the search from the current state to another state along an unexplored transition. The method returns a

boolean: whether the `forward` was successful. A forward fails, for example, if there are no unexplored transitions from the current state. The `forward` method also checks whether any property is violated after the unexplored transition has been traversed (we will come back to this later).

Our implementation of the search method is similar to the implementation of depth-first search by means of a stack (see, for example, [CLRS09, Chapter ?]. JPF keeps track of the stack and which transitions have already been traversed. Before the start of the `search` method JPF has already put the initial state on the stack. The methods `forward` and `backtrack`, also inherited from the `Search` class, are the JPF counterparts of push and pop, respectively. Note that the stack is empty if and only if a pop fails. In our setting this amounts to the `backtrack` failing, that is, returning false.

```
public void search() {
  do {
    while (this.forward()) {
      // do nothing
    }
  } while (this.backtrack());
}
```

To reduce the number of calls of the methods `forward` and `backtrack`, we use the following three methods from the `Search` class that categorize the current state. The method `isNewState` tests whether the current state has been visited before by JPF. The method `isEndState` tests whether the current state is a final state. The method `isIgnoredState` tests whether the current state should be ignored by the search. States can, for example, be ignored by using in the system under test the method `ignoreIf` of JPF's class `Verify` which is part of the package `gov.nasa.jpf.vm` (the `Verify` class willl be discussed in more detail in Chapter **??**). Several other methods that characterize the current state can be found in the `Search` class.

```
public void search() {
  do {
    while (this.forward() &&
           this.isNewState() &&
           !this.isEndState()) {
      // do nothing
    }
  } while (this.backtrack());
}
```

Whenever we reach an ignored state, we try to backtrack as well.

```
public void search() {
  do {
    while (this.forward() &&
           this.isNewState() &&
           !this.isEndState() &&
           !this.isIgnoredState()) {
      // do nothing
    }
  } while (this.backtrack());
}
```

Below, we extend the above search method in several ways by adding different, orthogonal, aspects to the search.

## 9.3 Other Components

Other components of JPF, such as listeners, can end a search by setting the attribute `done` of the class `Search` to true. This is reflected in the search method as follows.

```
public void search() {
  do {
    while (!this.done &&
           this.forward() &&
           this.isNewState() &&
           !this.isEndState() &&
           !this.isIgnoredState()) {
      // do nothing
    }
  } while (!this.done &&
           this.backtrack());
}
```

Other components of JPF can also request a search to backtrack by means of the method `requestBacktrack` of the class `Search`. This method simply sets a boolean attribute to true. The method `checkAndResetBacktrackRequest` of the class `Search` tests whether a backtrack has been requested and resets the attribute to false. Requests of backtracks can be addressed in our search method as follows.

```
public void search() {
  do {
    while (!this.done &&
           !this.checkAndResetBacktrackRequest() &&
           this.forward() &&
           this.isNewState() &&
           !this.isEndState() &&
           !this.isIgnoredState()) {
      // do nothing
    }
  } while (!this.done &&
           this.backtrack());
}
```

A search can be configured in several ways. Next, we will introduce the JPF properties relevant to a search.

## 9.4 Search Properties

Recall that JPF can be configured to limit the depth of the search by setting the JPF property `search.depth_limit`. The `Search` class contains the attribute `depth`, which is initialized to zero, that can be used to keep track of the depth of the search. It also provides the method `getDepthLimit` which returns the maximal allowed depth of the search.

We can keep track of the depth of the search as follows.

```
protected boolean forward() {
  boolean successful = super.forward();
  if (successful) {
    this.depth++;
  }
  return successful;
}

protected boolean backtrack() {
  boolean successful = super.backtrack();
  if (successful) {
```

```
    this.depth--;
  }
  return successful;
}
```

We can limit the depth of the search as follows.

```
private boolean checkDepthLimit() {
  return this.depth < this.getDepthLimit();
}

public void search) {
  do {
    while (!this.done &&
          !this.checkAndResetBacktrackRequest() &&
          this.forward() &&
          this.checkDepthLimit() &&
          this.isNewState() &&
          !this.isEndState() &&
          !this.isIgnoredState()) {
      // do nothing
    }
  } while (!this.done &&
          this.backtrack());
}
```

Recall that the JPF property `search.min_free` captures the minimal amount of memory, in bytes, that needs to remain free. The method `checkStateSpaceLimit` of the class `Search` checks whether the minimal amount of memory that should be left free is still available. We will use the attribute `done` to end the search if we run almost out of memory as follows.

```
public boolean checkStateSpaceLimit() {
  boolean available = super.checkStateSpaceLimit();
  if (!available) {
    this.done = true;
  }
  return available;
}

public void search() {
  do {
    while (!this.done &&
          !this.checkAndResetBacktrackRequest() &&
          this.forward() &&
          this.checkDepthLimit() &&
          this.isNewState() &&
          !this.isEndState() &&
          !this.isIgnoredState() &&
          this.checkStateSpaceLimit()) {
      // do nothing
    }
  } while (!this.done &&
          this.backtrack());
}
```

Recall that the JPF property `search.multiple_errors` tells us whether the search should report multiple errors (or just the first one). The `forward` method also checks whether any property is violated after the unexplored transition has been traversed. If a violation has been detected then the attribute `done` is set to true if and only if JPF has been configured to report at most one error. The method `hasPropertyTermination` of the class `Search` checks whether a violation was encountered during the last transition. The method returns true if and only if a violation was encountered and the attribute `done` is set to true. In the context below (that is, being invoked immediate after the `forward` method), the latter denotes that JPF has been configured to report at most one error. Furthermore, if `done` is set to false (which in this context denotes that JPF has been configured to report multiple errors), then the attribute requesting a backtrack is set to true.

```
public void search() {
  do {
    while (!this.done &&
          !this.checkAndResetBacktrackRequest() &&
          this.forward() &&
          this.checkDepthLimit() &&
          this.isNewState() &&
          !this.isEndState() &&
          !this.isIgnoredState() &&
          this.checkStateSpaceLimit() &&
          !this.hasPropertyTermination()) {
      // do nothing
    }
  } while (!this.done &&
          this.backtrack());
}
```

Assume that we are in a state that is new and neither final nor ignored and also suppose that the `forward` call returns true. We distinguish the following three cases.

1. If the `forward` method does not encounter a violation, then `hasPropertyTermination` returns false and, hence, the search continues.
2. If the `forward` method encounters a violation and `search.multiple_errors` is set to false, then the attribute `done` is set to true and, therefore, `hasPropertyTermination` returns true. Hence, the search terminates.
3. If the `forward` method encounters a violation and `search.multiple_errors` is set to true, then the attribute `done` is set to false and, therefore, `hasPropertyTermination` returns false and the attribute requesting a backtrack is set to true. Hence, the search continues. In the next iteration, the search attempts to backtrack since `checkAndResetBacktrackRequest` returns true.

## 9.5 Notifications

A search should also notify listeners of particular events. The `Search` class provides the following methods.

```
void notifyStateAdvanced()
void notifyStateBacktracked()
void notifyStateProcessed()

void notifySearchStarted()
void notifySearchFinished()

void notifyPropertyViolated()
void notifySearchConstraintHit(String)
```

Note that the methods above correspond to the methods of the interface `SearchListener`, which can be found in the package `gov.nasa.jpf.search`.

The first group contains methods that notify listeners of events related to the current state of the search. The method `notifyStateAdvanced` notifies the listeners that the current state has been reached as a result of a successful `forward` invocation. The method `notifyStateProcessed` notifies the listeners that the current state has been fully explored, that is, it has no unexplored outgoing transitions. The method `notifyStateBacktracked` notifies the listeners that the current state has been reached by means of a backtrack. The method `notifySearchStarted` notifies the listeners that the search has started and the method `notifySearchFinished` notifies the listeners that the search has finished. Below, we present the simplest extension of the basic search so that we can introduce all the above mentioned notifications.

```
protected boolean forward() {
  boolean successful = super.forward();
  if (successful) {
    this.notifyStateAdvanced();
  } else {
    this.notifyStateProcessed();
  }
  return successful;
}

protected boolean backtrack() {
  boolean successful = super.backtrack();
  if (successful) {
    this.notifyStateBacktracked();
  }
  return successful;
}

public void search() {
  this.notifySearchStarted();
  do {
    while (this.forward() &&
          this.isNewState() &&
          !this.isEndState()) {
      // do nothing
    }
  } while (this.backtrack());
  this.notifySearchFinished();
}
```

## 9.6 The Complete Search

We have left to adding the appropriate invocations of the `notifyPropertyViolated` and `notifySearchConstraintHit` methods. The method `notifyPropertyViolated` notifies the listeners that a violation has been encountered in the current state. Recall that in our setting the method `hasPropertyTermination` returns true if and only if a violation of a property has been detected and JPF has been configured to report at most one error. Hence, this method cannot be used to report violations of properties. Immediately after an invocation of the `forward` method, the attribute `currentError` of the class `Search` is null if and only if no violation has been detected. Therefore, we can use this attribute to determine whether we should report a violation of a property.

```
protected boolean forward() {
```

```
  boolean successful = super.forward();
  if (successful) {
    this.notifyStateAdvanced();
    if (this.currentError != null) {
      this.notifyPropertyViolated();
    }
  } else {
    this.notifyStateProcessed();
  }
  return successful;
}
```

As we already mentioned in Section **??**, JPF checks two constraints by default: the depth of the seach and the amount of remaining memory. The method `notifySearchConstraintHit` notifies the listeners that a constraint has been violated. Combining all the above and adding the appropriate invocations of the `notifySearchConstraintHit` methods, we arrive at the following.

```
protected boolean forward() {
  boolean successful = super.forward();
  if (successful) {
    this.notifyStateAdvanced();
    this.depth++;
    if (this.getCurrentError() != null) {
      this.notifyPropertyViolated();
    }
  } else {
    this.notifyStateProcessed();
  }
  return successful;
}

protected boolean backtrack() {
  boolean successful = super.backtrack();
  if (successful) {
    this.notifyStateBacktracked();
    this.depth--;
  }
  return successful;
}

public boolean checkStateSpaceLimit() {
  boolean available = super.checkStateSpaceLimit();
  if (!available) {
    this.done = true;
    this.notifySearchConstraintHit("memory limit reached: " + this.minFreeMemory);
  }
  return available;
}

private boolean checkDepthLimit() {
  boolean below = this.depth < this.getDepthLimit();
  if (!below) {
    this.notifySearchConstraintHit("depth limit reached: " + this.depth);
```

```
  }
  return below;
}

public void search() {
  this.notifySearchStarted();
  do {
    while (!this.done &&
          !this.checkAndResetBacktrackRequest() &&
          this.forward() &&
          this.checkDepthLimit() &&
          this.isNewState() &&
          !this.isEndState() &&
          !this.isIgnoredState() &&
          this.checkStateSpaceLimit() &&
          !this.hasPropertyTermination()) {
      // do nothing
    }
  } while (!this.done &&
          this.backtrack());
  this.notifySearchFinished();
}
```

Although the code of the above `search` method is different from the code of the `search` method of JPF's `DFSearch` class, which is part of the package `gov.nasa.jpf.search`, their behaviour is the same.

## 9.7 Breadth-First Search

As a second example, we implement BFS in a class named `BFSearch`. We start with extending the class `Search`.

```
import gov.nasa.jpf.Config;
import gov.nasa.jpf.vm.VM;
import gov.nasa.jpf.search.Search;

public class BFSearch extends Search {
  public BFSearch(Config config, VM vm) {
    super(config, vm);
  }
}
```

To implement the basic search, we need a few new ingredients. In particular, we use two methods of the class `VM` which represents JPF's virtual machine. The method `getRestorableState` returns a `RestorableVMState` object which represents the current state. The class `RestorableVMState` is part of the package `gov.nasa.jpf.vm`. The method `restoreState(RestorableVMState)` restores the given state, that is, the current state becomes a previously visited state provided as an argument.

The class `Search` contains the method `supportRestoreState` that checks whether the search supports restoring states that have been stored. In the `Search` class it returns false. Here, we support restoring states and, hence, we override the method. With these ingredients we can build the following basic breadth-first search.

```
private RestorableVMState getRestorableState() {
  return this.getVM().getRestorableState();
}
```

```
private void restoreState(RestorableVMState state) {
  this.getVM().restoreState(state);
}

public boolean supportsRestoreState() {
  return true;
}

public void search() {
  Queue<RestorableVMState> queue = new LinkedList<RestorableVMState>();
  queue.offer(this.getRestorableState());
  while (!queue.isEmpty()) {
    RestorableVMState state = queue.poll();
    this.restoreState(state);
    while (this.forward()) {
      if (this.isNewState() &&
          !this.isEndState()) {
        queue.offer(this.getRestorableState());
      }
      this.backtrack();
    }
  }
}
```

Recall that the method `isNewState` of the class `Search` tests whether the current state is new, that is, it has not been visited before by the search. The method `isEndState` of the class `Search` tests whether the current state is an end state, that is, a final state. To ensure that the search terminates, we only enqueue new states. There is no need to enqueue final states.

```
public void search() {
  Queue<RestorableVMState> queue = new LinkedList<RestorableVMState>();
  queue.offer(this.getRestorableState());
  while (!queue.isEmpty()) {
    RestorableVMState state = queue.poll();
    this.restoreState(state);
    while (this.forward()) {
      if (this.isNewState() &&
          !this.isEndState()) {
        queue.offer(this.getRestorableState());
      }
      this.backtrack();
    }
  }
}
```

We do not store ignored states in the queue.

```
public void search() {
  Queue<RestorableVMState> queue = new LinkedList<RestorableVMState>();
  queue.offer(this.getRestorableState());
  while (!queue.isEmpty()) {
    RestorableVMState state = queue.poll();
    this.restoreState(state);
    while (this.forward()) {
```

```
      if (this.isNewState() &&
          !this.isEndState() &&
          !this.isIgnoredState()) {
        queue.offer(this.getRestorableState());
      }
      this.backtrack();
    }
  }
}
```

As before, the attribute `done` is used to end the search.

```
public void search() {
  Queue<RestorableVMState> queue = new LinkedList<RestorableVMState>();
  queue.offer(this.getRestorableState());
  while (!this.done &&
         !queue.isEmpty()) {
    RestorableVMState state = queue.poll();
    this.restoreState(state);
    while (!this.done &&
           this.forward()) {
      if (this.isNewState() &&
          !this.isEndState() &&
          !this.isIgnoredState()) {
        queue.offer(this.getRestorableState());
      }
      this.backtrack();
    }
  }
}
```

In our BFS implementation we decided *not* to support backtrack requests by other JPF components. The class `Search` contains the method `supportBacktrack` which tests whether a search supports backtrack requests. This method of the `Search` class always returns true. In our subclass `BFSearch`, we override this method as follows.

```
public boolean supportBacktrack() {
  return false;
}
```

As is well known, breadth-first search visits the states level by level. Hence, we only need to increment the depth whenever we reach the end of a level. To keep track of the end of a level, we use the value `null`. After enqueuing the initial state in line 3, we enqueue `null` in line 4. Whenever we dequeue `null`, that is, we reach the end of a level, we increment the depth and enqueue `null` to keep track of the end of the next level (see line 7–10). Note that the loop of line 5-23 maintains the invariant that the queue contains `null`. Hence, we replace the condition `!queue.isEmpty()` with `queue > 1` in line 6.

```
1 public void search() {
2   Queue<RestorableVMState> queue = new LinkedList<RestorableVMState>();
3   queue.offer(this.getRestorableState());
4   queue.offer(null);
5   while (!this.done &&
6          queue.size() > 1) {
7     RestorableVMState state = queue.poll();
8     if (state == null) {
9       this.depth++;
```

```
10        queue.offer(null);
11      } else {
12        this.restoreState(state);
13        while (!this.done &&
14               this.forward()) {
15          if (this.isNewState() &&
16              !this.isEndState() &&
17              !this.isIgnoredState()) {
18            queue.offer(this.getRestorableState());
19          }
20          this.backtrack();
21        }
22      }
23    }
24  }
```

As for the depth-first search, we introduce a method `checkDepthLimit` that checks whether the depth has not reached the limit yet. Since the depth is incremented in the outer loop, we use that method in line 11.

```
1  private boolean checkDepthLimit() {
2    return this.depth < this.getDepthLimit();
3  }
4
5  public void search() {
6    Queue<RestorableVMState> queue = new LinkedList<RestorableVMState>();
7    queue.offer(this.getRestorableState());
8    queue.offer(null);
9    while (!this.done &&
10          queue.size() > 1 &&
11          this.checkDepthLimit()) {
12     RestorableVMState state = queue.poll();
13     if (state == null) {
14       this.depth++;
15       queue.offer(null);
16     } else {
17       this.restoreState(state);
18       while (!this.done &&
19              this.forward()) {
20         if (this.isNewState() &&
21             !this.isEndState() &&
22             !this.isIgnoredState()) {
23           queue.offer(this.getRestorableState());
24         }
25         this.backtrack();
26       }
27     }
28   }
29 }
```

To end the search when insufficient memory is available, we use the method `checkStateSpaceLimit` and the attribute `done` as follows.

```
public boolean checkStateSpaceLimit() {
  boolean available = super.checkStateSpaceLimit();
```

```
  if (!available) {
    this.done = true;
  }
  return available;
}

public void search() {
  Queue<RestorableVMState> queue = new LinkedList<RestorableVMState>();
  queue.offer(this.getRestorableState());
  queue.offer(null);
  while (!this.done &&
       queue.size() > 1 &&
       this.checkDepthLimit()) {
    RestorableVMState state = queue.poll();
    if (state == null) {
      this.depth++;
      queue.offer(null);
    } else {
      this.restoreState(state);
      while (!this.done &&
           this.forward() &&
           this.checkStateSpaceLimit()) {
        if (this.isNewState() &&
           !this.isEndState() &&
           !this.isIgnoredState()) {
          queue.offer(this.getRestorableState());
        }
        this.backtrack();
      }
    }
  }
}
```

The JPF property `search.multiple_errors` can be dealt with similarly as in our implementation of depth-first search. Recall that `forward` method checks whether any property is violated after the unexplored transition has been traversed. If a violation has been detected then the attribute `done` is set to true if and only if JPF has been configured to report at most one error. The method `hasPropertyTermination` of the class `Search` checks whether a violation was encountered during the last transition. The method returns true if and only if a violation was encountered and the attribute `done` is set to true. Therefore, we add an invocation of the method `hasPropertyTermination` on line 17, that is, being invoked immediate after the `forward` method. Furthermore, if a violation was detected then we should not enqueue the resulting state. The method `isErrorState` checks whether a violation has occurred in the current state. We add it in line 21.

```
1  public void search() {
2    Queue<RestorableVMState> queue = new LinkedList<RestorableVMState>();
3    queue.offer(this.getRestorableState());
4    queue.offer(null);
5    while (!this.done &&
6         queue.size() > 1 &&
7         this.checkDepthLimit()) {
8      RestorableVMState state = queue.poll();
9      if (state == null) {
10       this.depth++;
```

```
11        queue.offer(null);
12      } else {
13        this.restoreState(state);
14        while (!this.done &&
15               this.forward() &&
16               !this.checkStateSpaceLimit() &&
17               !this.hasPropertyTermination()) {
18          if (this.isNewState() &&
19              !this.isEndState() &&
20              !this.isIgnoredState() &&
21              !this.isErrorState()) {
22            queue.offer(this.getRestorableState());
23          }
24          this.backtrack();
25        }
26      }
27    }
28  }
```

Assume that the `forward` call returns true and that we end up in a state that is new and neither final nor ignored. We distinguish the following three cases.

1. If the `forward` method does not encounter a violation, then `hasPropertyTermination` returns false and, hence, the search continues.
2. If the `forward` method encounters a violation and `search.multiple_errors` is set to false, then the attribute `done` is set to true and, therefore, `hasPropertyTermination` returns true. Hence, the search terminates.
3. If the `forward` method encounters a violation and `search.multiple_errors` is set to true, then the attribute `done` is set to false and, therefore, `hasPropertyTermination` returns false. Hence, the search continues. Since the state is an error state, it is not enqueued.

In the notification code, we use two methods that we have not discussed before. The method `notifyStateStored` notifies the listeners that the current state has been stored (so that it can be restored later). The method `notifyStateRestored` notifies the listeners that the current state has been restored (by means of the `restoreState` method). Below, we introduce all the notifications.

```
protected boolean forward() {
  boolean successful = super.forward();
  if (successful) {
    this.notifyStateAdvanced();
    if (this.getCurrentError() != null) {
      this.notifyPropertyViolated();
    }
  } else {
    this.notifyStateProcessed();
  }
  return successful;
}

protected boolean backtrack() {
  boolean successful = super.backtrack();
  if (successful) {
    this.notifyStateBacktracked();
```

```
  }
  return successful;
}

private boolean checkDepthLimit() {
  boolean below = this.depth < this.getDepthLimit();
  if (!below) {
    this.notifySearchConstraintHit("depth limit reached: " + this.depth);
  }
  return below;
}

public boolean checkStateSpaceLimit() {
  boolean available = super.checkStateSpaceLimit();
  if (!available) {
    this.done = true;
    this.notifySearchConstraintHit("memory limit reached: " + this.minFreeMemory);
  }
  return available;
}

public void search() {
  this.notifySearchStarted();
  Queue<RestorableVMState> queue = new LinkedList<RestorableVMState>();
  queue.offer(this.getRestorableState());
  this.notifyStateStored();
  queue.offer(null);
  while (!this.done &&
       queue.size() > 1 &&
       this.checkDepthLimit()) {
    RestorableVMState state = queue.poll();
    if (state == null) {
      this.depth++;
      queue.offer(null);
    } else {
      this.restoreState(state);
      this.notifyStateRestored();
      while (!this.done &&
           this.forward() &&
           !this.checkStateSpaceLimit() &&
           !this.hasPropertyTermination()) {
        if (this.isNewState() &&
           !this.isEndState() &&
           !this.isIgnoredState() &&
           !this.isErrorState()) {
          queue.offer(this.getRestorableState());
          this.notifyStateStored();
        }
      this.backtrack();
      }
    }
  }
```

```
  this.notifySearchFinished();
}
```

The code of the above `search` method is quite different from the code of the `search` method of the class `BFSHeuristic` of the package `gov.nasa.jpf.search.heuristic`. There are also the three differences in behaviour. First of all, our `search` method checks whether the state space limit has been reached, whereas JPF's does not. Secondly, JPF's `search` method signals that the initial state has been stored before it signals that the search has started, whereas our `search` method signals those events in the opposite order. Thirdly, JPF's `search` method does not signal that the initial state has been restored (as it documented in the code), whereas our `search` method does.

# Chapter 10

# Handling Native Methods

Every JVM includes a native method interface (JNI) that allows Java applications to invoke native methods, that is, methods that are implemented in a language other than Java but that are invoked from the Java application). This feature allows programmers to use code that has been already implemented in other languages. In some cases, accessing code such as C and C++ from applications written in Java can increase the performance. Another advantage of JNI is that it can be used when Java does not support certain platform-dependent features. Many of the classes of the Java standard library include invocations of native code.

The core of the JPF is a JVM that is able to execute all of the bytecode instructions that are created by a Java compiler. JPF itself is written in the Java programming language. That means that JPF is running on top of another JVM which we call it the host JVM. Figure 10 demonstrates the different layers that are involved in model checking a system under test using JPF.

Since the core of the JPF is a JVM that executes only Java bytecode instructions, it is not able to execute native methods. Consider, for example, the following application.

```
public class Sine {
  public static void main(String[] args) {
    System.out.println(StrictMath.sin(0.3));
  }
}
```

If we verify the above code, JPF reports the following error.

```
====================================================== error 1
gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
java.lang.UnsatisfiedLinkError: cannot find native java.lang.StrictMath.sin
at java.lang.StrictMath.sin(no peer)
at Sinus.main(Sine.java:3)
```

Its verification effort fails as it encounters the native method `sin` of the class `StrictMath`. However, JPF provides two different mechanisms, which can be combined, to handle calls to native methods. We will discuss these in the next sections. We will also present an extension of JPF, called `jpf-nhandler`[1]. It automates the use of both mechanisms to handle calls to native methods. The only thing left to the developer is the application configuration file.

### 10.0.1 Peer Classes

One way to handle native calls in JPF is using peer classes. These are the special classes that are considered as part of the system under test. The peer classes are verified by JPF and they are completely unknown to the host JVM. By

[1] `bitbucket.org/nastaran/jpf-nhandler`

implementing peer classes, we force JPF to use an alternative version of certain Java classes instead of the original ones. More specifically, if there exists a peer class for some class, JPF loads and uses the peer class instead of the class itself. Therefore, we can make JPF not verify certain classes, and use the peer classes as alternatives.

Recall that the `StrictMath` class includes the `sin` method, which is defined as native. One way handle the `sin` native call is to create a peer class `StrictMath` that implements the `sin` method in pure Java. Since the original class `StrictMath` is part of the package `java.lang`, our peer class is part of that package as well. To implement the `sin` method in pure Java, we can use, for example, Bhaskara I's sine approximation formula as follows.

```
package java.lang;

public class StrictMath {
  public static double sin (double a) {
    return 16 * a * (Math.PI - a) / (5 * Math.PI * Math.PI - 4 * a * (Math.PI - a));
  }
}
```

Note that we do not need to implement all methods of the `StrictMath` class. After we have implemented this peer class, we still have to ensure that JPF never loads and verifies the standard class `StrictMath`. This is accomplished by adding the peer class `StrictMath` to JPF's classpath.

```
target=Sine
classpath=C:/Users/franck/workspace/examples/bin
```

In this case, the directory `C:\Users\franck\workspace\examples\bin\java\lang` should contains the file `StrictMath.class`.

## 10.1 Native Peer Classes

JPF's model Java interface (MJI) can be used to transfer the execution from JPF to the host JVM. The so called native peer classes play a key role in MJI. JPF uses a specific name pattern to associate the native peer classes and their methods with the corresponding classes and methods. For example, the native peer class associated with `sun.misc.Unsafe` is named `JPF_sun_misc_Unsafe` (see Figure **??**). Whenever JPF gets to a call associated with a native peer method, it delegates the call to the host JVM. Hence, the native call is not model checked, which is impossible since JPF can only handle Java bytecode, but executed on the host JVM. Great care has to be taken when developing a native peer class. For example, since classes and objects are represented differently in JPF than in an ordinary JVM, in a native peer class one often has to translate from the one representation to the other and back.

# Chapter 11

# Testing JPF Components

In order to test components of JPF, such as listeners, search strategies and reporters, JPF provides a testing framework similar to JUnit. We assume that the reader is already familiar with JUnit. For an introduction to Junit, we refer the reader to junit.org/junit4.

Before we introduce JPF's testing framework, let us first consider the two components that are needed to test that JPF can detect failed assertions. We need an app with an assertion that fails and an application properties file. As app, we can use, for example, the following.

```
public class FailedAssertion {
  public static void main(String[] args) {
    assert false;
  }
}
```

To run JPF on this app, we create the following application properties file.

```
target=FailedAssertion
classpath=.
```

If we run JPF on the above application properties file, it produces the following output.

```
JavaPathfinder core system v8.0 (rev 26e11d1de726c19ba8ae10551e048ec0823aabc6) - (C) 2005-2


====================================================== system under test
FailedAssertion.main()

====================================================== search started: 2/24/20 4:08 PM

====================================================== error 1
gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
java.lang.AssertionError
      at FailedAssertion.main(FailedAssertion.java:3)


====================================================== snapshot #1
thread java.lang.Thread:{id:0,name:main,status:RUNNING,priority:5,isDaemon:false,lockCount
  call stack:
      at FailedAssertion.main(FailedAssertion.java:3)
```

```
====================================================== results
error #1: gov.nasa.jpf.vm.NoUncaughtExceptionsProperty "java.lang.AssertionError
at FailedAssertion.main(..."

====================================================== statistics
elapsed time:  00:00:00
states:        new=1,visited=0,backtracked=0,end=0
search:        maxDepth=1,constraints=0
choice generators: thread=1 (signal=0,lock=1,sharedRef=0,threadApi=0,reschedule=0), data=0
heap:          new=366,released=0,maxLive=0,gcCycles=0
instructions:  3207
max memory:    57MB
loaded code:   classes=65,methods=1371

====================================================== search finished: 2/24/20 4:08 PM
```

From the above output we can conclude that JPF indeed detects the failed assertion in the app. Checking the output produced by JPF manually is cumbersome. Having the app and the corresponding properties in two different files introduces the risk that the files get out of sync when changing either file. JPF's framework addresses both shortcomings: it allows us to check the result of the test case programmatically and it also combines the app and its properties into a single file.

## 11.1 A Simple Example

Before discussing the details of JPF's testing framework, let us first recast the above test in the framework. The abstract class `TestJPF`, which is part of the package `gov.nasa.jpf.util.test`, is the heart of JPF's framework. To implement a test, we extend this class using the following skeleton.

```
import gov.nasa.jpf.util.test.TestJPF;
import org.junit.Test;

public class ...Test extends TestJPF {

  @Test
  public void test...() {
    ...
  }
}
```

To test that JPF detects failed assertions, we combine the above app and the corresponding properties file into a single test method as follows.

```
1 public class AssertionTest extends TestJPF {
2
3   @Test
4   public void testAssertFails() {
5     if (this.verifyAssertionError("+classpath=.")) {
6       assert false;
7     }
8   }
9 }
```

The method `verifyAssertionError` takes as argument the content of the application properties file. Note that these properties are provided in the same format as when given to JPF as command line arguments, that is, each property is prefixed by a `+`. Line 6 contains the body on the `main` method of the app.

To test that a successful assertion is handled properly by JPF, we can add, for example, the following test method to the above class.

```
@Test
public void testAssertSucceeds() {
  if (this.verifyNoPropertyViolation("+classpath=.")) {
    assert true;
  }
}
```

## 11.2 Running a JPF Test

A JPF test can be run as an ordinary JUnit test in Eclipse. For convenience, we add a `main` method, as shown below, so that the JPF test can also be run as an ordinary app. This app takes the names of the test methods to be run as command line arguments. If no command line arguments are provided, then all tests are run.

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

When we run the above JPF test as an ordinary app, it produces the following output.

```
..................................... testing testAssertSucceeds()
  running jpf with args: +classpath=.
JavaPathfinder core system v8.0 (rev 32+) - (C) 2005-2014 United States Government. All ri


====================================================== system under test
AssertionTest.runTestMethod()

====================================================== search started: 2/25/20 4:25 PM

====================================================== results
no errors detected

====================================================== search finished: 2/25/20 4:25 PM
..................................... testAssertSucceeds: Ok

..................................... testing testAssertFails()
  running jpf with args: +classpath=.
JavaPathfinder core system v8.0 (rev 32+) - (C) 2005-2014 United States Government. All ri


====================================================== system under test
AssertionTest.runTestMethod()

====================================================== search started: 2/25/20 4:25 PM

====================================================== error 1
gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
```

```
java.lang.AssertionError
      at AssertionTest.testAssertFails(test/BasicTest.java:25)
      at java.lang.reflect.Method.invoke(gov.nasa.jpf.vm.JPF_java_lang_reflect_Method)
      at gov.nasa.jpf.util.test.TestJPF.runTestMethod(gov/nasa/jpf/util/test/TestJPF.java:


====================================================== snapshot #1
thread java.lang.Thread:{id:0,name:main,status:RUNNING,priority:5,isDaemon:false,lockCount
  call stack:
      at gov.nasa.jpf.util.test.TestJPF.runTestMethod(TestJPF.java:650)


====================================================== results
error #1: gov.nasa.jpf.vm.NoUncaughtExceptionsProperty "java.lang.AssertionError
at AssertionTest.testAs..."

====================================================== search finished: 2/25/20 4:25 PM
.......................................... testAssertFails: Ok

.......................................... execution of testsuite: AssertionTest SUCCEEDED
.... [1] testAssertSucceeds: Ok
.... [2] testAssertFails: Ok
.......................................... tests: 2, failures: 0, errors: 0
```

Now that we have developed and run a simple example of a JPF test, let us dive into the details.

## 11.3 The `verifyNoPropertyViolation` Method

We have already seen the `verifyNoPropertyViolation` method in the above simple example. The method takes as argument an array of strings. These are provided to JPF as command line arguments. Usually, these are the properties one would put in the application properties file. However, one can also use command line arguments such as `-log` and `-show` as discussed in Section **??**. The method expects that no property is violated when JPF is run on the test method with the given command line arguments. If, however, a property violation is encountered by JPF, then the test fails.

Recall that JPF is a virtual machine and that JPF, since it is implemented in Java, runs on a Java virtual machine. We call the latter the host JVM. When a JPF test method is run, it is both executed by the host JVM and model checked by JPF, that is, executed by JPF's virtual machine. Consider the following test method.

```
1 private static final String PROPERTIES = { "+classpath=." };
2
3 @Test
4 public void test() {
5   System.out.println("1");
6   System.out.println(this.verifyNoPropertyViolation(PROPERTIES));
7   System.out.println("2");
8 }
```

When the above test is run, it is executed by the host JVM. In line 6, the host JVM invokes the `verifyNoPropertyViolation` method. As a result, JPF is invoked on the test method itself with the argument of `verifyNoPropertyViolation` as its command line arguments. Subsequently, JPF model checks the test method. In line 6, JPF's virtual machine invokes the `verifyNoPropertyViolation` method. This method invocation simply returns true. Once JPF has model checked the test method, the host JVM continues its execution of the test method with the invocation of the method `verifyNoPropertyViolation`. This time the method returns false. (For the curious reader, the fact

that the method `verifyNoPropertyViolation` behaves differently when being executed by the host JVM from being executed by JPF's virtual machine is accomplished by introducing a native peer class.) Subsequently, the host JVM executes the remainder of the test method. The above test method, executed as a JUnit test, is successful and gives rise to the following output.

```
1
  running jpf with args: +classpath=.
JavaPathfinder core system v8.0 (rev 32+) - (C) 2005-2014 United States Government. All ri


============================================================ system under test
SampleTest.runTestMethod()

============================================================ search started: 2/25/20 4:48 PM
1
true
2

============================================================ results
no errors detected

============================================================ search finished: 2/25/20 4:48 PM
false
2
```

The host JVM first executes line 5 of the test method, resulting in the output on line 1. Next, JPF is run on the test method, giving rise to line 2–17 of the output. Note that the invocation of the `verifyNoPropertyViolation` method by JPF returns true, as can be seen on line 11 of the output. Subsequently, the host JVM executes the remainder of the test method: the `verifyNoPropertyViolation` method invocation returns false, which can be seen at line 18 of the output, and finally 2 is printed.

Although the `verifyNoPropertyViolation` method can be invoked multiple times in a test method, resulting in JPF being run multiple times, we will only include a single invocation of the `verifyNoPropertyViolation` method in our test methods.

Let us revisit the test that checks if a successful assertion is handled properly by JPF.

```
@Test
public void testAssertSucceeds() {
  if (this.verifyNoPropertyViolation(PROPERTIES)) {
    assert true;
  }
}
```

When executing the test `testAssertSucceeds`, the host JVM first invokes the `verifyNoPropertyViolation` method. As a result, JPF is run on the `testAssertSucceeds` method. When JPF invokes the `verifyNoPropertyViolation` method, the method returns true. As a result, JPF executes, that is, model checks, also line 4. Once JPF has model checked the `testAssertSucceeds` method, the host JVM returns from the invocation of the `verifyNoPropertyViolation` method. Since the return is false this time, the host JVM does not execute line 4 of the `testAssertSucceeds` method. Although preventing the host JVM from executing this line of code does not impact the test, this pattern will be very helpful in cases we will discuss next.

## 11.4 The `verifyAssertionError` Method

The method `verifyAssertionError` also takes an array of strings as argument. Again, these are provided to JPF as command line arguments. The method expects that JPF encounters an assertion error when run on the test method.

If JPF does not encounter such an error, the test fails.

Consider the following test method.

```
1 @Test
2 public void test() {
3   System.out.println("1");
4   System.out.println(this.verifyAssertionError(PROPERTIES));
5   System.out.println("2");
6 }
```

The above test method, executed as a JUnit test, fails and produces the following output.

```
1 1
2 running jpf with args: +classpath=/courses/4315/workspace/4315/bin/
3 JavaPathfinder core system v8.0 (rev 32+) - (C) 2005-2014 United States Government. All ri
4
5
6 ============================================================ system under test
7 SampleTest.runTestMethod()
8
9 ============================================================ search started: 2/25/20 9:10 PM
10 1
11 true
12 2
13
14 ============================================================ results
15 no errors detected
16
17 ============================================================ search finished: 2/25/20 9:10 PM
```

The host JVM first executes line 3 of the test method, resulting in the output on line 1. Next, JPF is run on the test method, giving rise to line 2–17 of the output. Note that the invocation of the `verifyAssertionError` method by JPF returns true, as can be seen on line 11 of the output. Subsequently, the host JVM executes the remainder of the test method: the `verifyAssertionError` causes the test to fail, because no assertion error was detected while model checking the test method. Hence, no further output is produced.

Let us return to the following test method that we have already seen earlier.

```
1 @Test
2 public void testAssertFails() {
3   if (this.verifyAssertionError("+classpath=.")) {
4     assert false;
5   }
6 }
```

When executing the test `testAssertFails`, the host JVM first invokes the `verifyAssertionError` method. As a result, JPF is run on the `testAssertFails` method. When JPF invokes the `verifyAssertionError` method, the method returns true. As a result, JPF executes, that is, model checks, also line 4, and detects an assertion error. Once JPF has model checked the `testAssertFails` method, the host JVM returns from the invocation of the `verifyAssertionError` method. Since the return is false this time, the host JVM does not execute line 4 of the `testAssertFails` method. In this example, it is essential that the host JVM does not execute line 4. If it were to execute that line, the test would fail.

## 11.5 The `verifyUnhandledException` Method

The method `verifyUnhandledException` expects JPF to detect an unhandled exception. It takes two arguments: the name of the exception class and the command line arguments to be provided to JPF. To illustrate this method, consider the following very simple test method.

```
1 @Test
2 public void testUnhandledException() {
3   if (this.verifyUnhandledException("java.lang.RuntimeException", PROPERTIES)) {
4     throw new RuntimeException();
5   }
6 }
```

When executing the test `testUnhandledException`, the host JVM first invokes the `verifyUnhandledException` method. As a result, JPF is run on the `testUnhandledException` method. When JPF invokes the `verifyUnhandledExceptio` method, the method returns true. As a result, JPF executes, that is, model checks, also line 4, and detects an unhandled exception. Once JPF has model checked the `testUnhandledException` method, the host JVM returns from the invocation of the `verifyUnhandledException` method. Since the return is false this time, the host JVM does not execute line 4 of the `testUnhandledException` method. Also in this example, it is essential that the host JVM does not execute line 4.

One has provided the fully qualified name of the exception class. If we were to use the string `"RuntimeException"` instead of `"java.lang.RuntimeException"` in line 3, then the test would fail. One cannot use the fully qualified name of a superclass either. For example, if we were to use `"java.lang.Exception"` instead in line 3, the test would fail as well.

## 11.6 The `verifyPropertyViolation` Method

To test that JPF detects a property violation we use the method `verifyPropertyViolation`. This method takes two arguments: the type of the property that is expected to be violated and, as usual, the command line arguments provided to JPF. To specify the type of property, we use the class `TypeRef`, which is part of the package `gov.nasa.jpf.util`. We can test that JPF detects that an exception has not been caught as follows.

```
@Test
public void testPropertyViolation() throws Exception {
  TypeRef type = new TypeRef("gov.nasa.jpf.vm.NoUncaughtExceptionsProperty");
  if (this.verifyPropertyViolation(type, PROPERTIES)) {
    throw new Exception();
  }
}
```

Note that the test method `testPropertyViolation` specifies that it may throw an `Exception`. Without this specification, the test method does not compile.

## 11.7 The `isJPFRun` and `isJUnitRun` Methods

# Bibliography


[BBF+01] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, Philippe Schnoebelen, and Pierre McKenzie. *Systems and software verification: Model-checking techniques and tools*. Springer-Verlag, 2001.

[BJC+13] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. Reversible debugging software. Technical report, Cambridge University, Cambridge, United Kingdom, January 2013.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. The MIT Press, 2008.

[CDH+00] James Corbett, Matthew Dwyer, John Hatcliff, Shawn Laubach, Corina Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code. In Carlo Ghezzi, Mehdi Jazayeri, and Alexander Wolf, editors, *Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448, Limerick, Ireland, June 2000. ACM.

[CGK+18] Edmund Clarke, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. *Model checking*. The MIT Press, 2018.

[CLRS09] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to algorithms*. The MIT Press, 2009.

[Kra18] Herb Krasner. The cost of poor quality software in the US: A 2018 report. Technical report, Consortium for IT Software Quality, Needham, MA, USA, September 2018.

[RDH03] Robby, Matthew Dwyer, and John Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 267–276, Helsinki, Finland, September 2003. ACM.

[VHB+03] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.

[VHBP00] Willem Visser, Klaus Havelund, Guillaume Brat, and Seungjoon Park. Model checking programs. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, pages 3–12, Grenoble, France, September 2000. IEEE.