



Quick answers to common problems

OpenJDK Cookbook

Over 80 recipes to build and extend your very own version of Java platform using OpenJDK project

Alex Kasko
Alexey Mironchenko

Stanislav Kobylanskiy

[PACKT]
PUBLISHING

OpenJDK Cookbook

Over 80 recipes to build and extend your very own version of Java platform using OpenJDK project

Alex Kasko

Stanislav Kobylanskiy

Alexey Mironchenko



BIRMINGHAM - MUMBAI

```
echo The JAVA_HOME environment variable is not defined correctly
exit /b 1

:noJnlp
echo Please provide JNLP file as first argument
exit /b 1
```

4. Run the application using the launcher script:

```
wslauncher.bat dynamictree_webstart.jnlp
```

How it works...

The `netx.jar` module must be added to the boot classpath as it cannot be run directly because of security reasons.

The `wslauncher.bat` script tries to find the Java executable using the `JAVA_HOME` environment variable and then launches specified JNLP through `netx.jar`.

There's more...

The `wslauncher.bat` script may be registered as a default application to run the JNLP files. This will allow you to run WebStart applications from the web browser. But the current script will show the batch window for a short period of time before launching the application. It also does not support looking for Java executables in the Windows Registry. A more advanced script without those problems may be written using Visual Basic script (or any other native scripting solution) or as a native executable launcher. Such solutions lay down the scope of this book.

See also

- ▶ The *Preparing the IcedTea Java WebStart implementation for Mac OS X* recipe
- ▶ The *Building 64-bit OpenJDK 7 on Windows 7 x64 SP1* recipe from *Chapter 3, Building OpenJDK 7*
- ▶ The IcedTea-Web project website at <http://icedtea.classpath.org/wiki/IcedTea-Web>
- ▶ The NetX project website at <http://jnlp.sourceforge.net/netx/>
- ▶ The Java WebStart developers guide at <http://docs.oracle.com/javase/6/docs/technotes/guides/javaws/>
- ▶ Articles on ActiveX technology from Microsoft that is similar to NPAPI at <http://msdn.microsoft.com/en-us/library/aa751968%28v=vs.85%29.aspx>

8

Hacking OpenJDK

In this chapter we will cover:

- ▶ Setting up the development environment with NetBeans
- ▶ Working with Mercurial forests
- ▶ Understanding OpenJDK 6 and 7 incremental builds
- ▶ Debugging Java code using NetBeans
- ▶ Debugging C++ code using NetBeans
- ▶ Using NetBeans to compile HotSpot
- ▶ Using HotSpot dev parameters
- ▶ Adding new intrinsic to HotSpot
- ▶ Building VisualVM from the source code
- ▶ Creating a plugin for VisualVM
- ▶ Getting benefits from the AdoptOpenJDK project

Introduction

The real beauty of OpenJDK is its open nature, which means that developers can not only use it to run an application, but also change it to their needs or contribute to its development. The availability of the source code and ease of access to it opens huge opportunities to individuals who have special requirements, or just want to learn more about the way JVM works internally and want to adapt it for any special requirements. This chapter will help you to get into it and provide some recipes to make the process of setting up the required development environment as easy as possible.

At first, it will cover how to set up the development environment and which tools are required to get started. It will cover the IDE setup, and some tweaks required to launch JVM and start debugging. The next step is to make changes and rebuild the code, and the latter is going to be slightly different from the normal build described in *Chapter 5, Building IcedTea*, *Chapter 6, Building IcedTea with Other VM Implementations*, and *Chapter 7, Working with WebStart and the Browser Plugin*. The rest of that section dedicated to will be useful techniques which can be used to debug changes.

This chapter assumes that the reader has a reasonable knowledge of C++ and Java. Any knowledge of JVM is ideal, as the reader should know what JIT is and how it works.

Most of the recipes in the JIT section are independent and can be executed separately, so the reader just can pick what he/she needs and proceed with it.

Setting up the development environment with NetBeans

This recipe will cover the steps required to install, run, and set up the project in NetBeans IDE. NetBeans is an open source IDE for developing, primarily in Java. It also has rich support for C++ and that makes it a good tool for OpenJDK development which uses both languages. This recipe uses NetBeans IDE v.7.

Getting ready

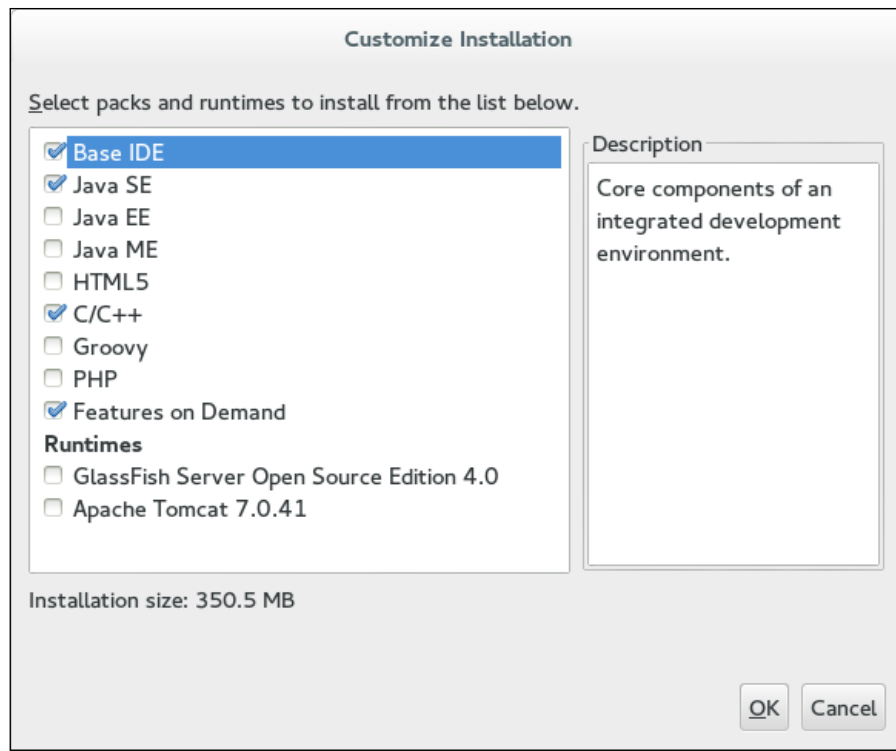
Download the latest version of the NetBeans All bundle for your platform from <http://www.netbeans.org/downloads>. The All bundle must have C/C++ and Java support in the same IDE. It is also necessary to have the OpenJDK code checked out and available on the machine.

Make sure that everything is set up for the OpenJDK build and it can be executed without errors. How to do that is described in *Chapter 2, Building OpenJDK 6*, *Chapter 3, Building OpenJDK 7*, and *Chapter 4, Building OpenJDK 8*.

How to do it...

We will install and configure the NetBeans IDE that is used in the OpenJDK project as the standard one.

1. First we need to install the NetBeans IDE. This is a very simple process which consists of a few simple steps. Run the downloaded executable and at the bottom of the first screen, select the **Customize** button. This will show following window:



Ensure that **Base IDE**, **Java SE**, **C/C++**, and **Features on Demand** are selected. The rest are optional and not required to run and debug OpenJDK, but there is no harm in installing that functionality.

2. After the setup, all plugins you should update to the latest version. Updates are available via the **Help/Check for updates** menu item.
3. When NetBeans is set up, it is required to make a slight change to its configuration.

OpenJDK is a big project and its memory requirements are bigger than the ones defined in the default settings. To increase the memory available for IDE:

1. Go to the `$HOME/.netbeans/NETBEANS_VERSION/etc` folder (on Windows `$HOME` is `%HOMEPATH%`).
2. If the folder doesn't exist, create it.

- Then, if that folder doesn't have the `netbeans.conf` file, copy it from the Netbeans installation directory, which is located in the `etc` folder.
- Open the file with any text editor and locate the `netbeans_default_options` parameter which it should look similar to this:

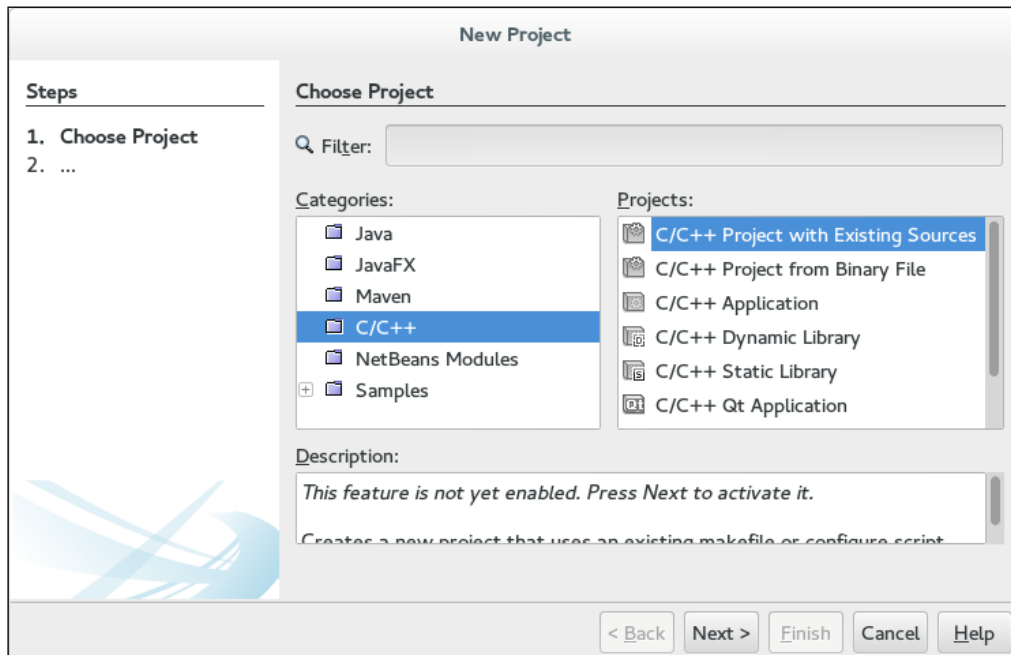
```
netbeans_default_options="-J-client -J-Xss2m -J-Xms32m
-J-XX:PermSize=32m -J-Dapple.laf.useScreenMenuBar=true
-J-Dapple.awt.graphics.UseQuartz=true -J-Dsun.java2d.
noddraw=true -J-Dsun.java2d.dpiaware=true -J-Dsun.zip.
disableMemoryMapping=true"
```

- When the parameter is located, add `-J-Xmx2g`, or, if that option is already present, update it to a value not less than 2G (2 gigabytes). This will increase the memory available to JDK to 2G. Restart your IDE if it was running before in order to apply that change.




It is worth adding that, because of the large memory requirement of Netbeans IDE, It's recommended to run it on a system that is capable of providing nothing less than 2 GB of memory to the process. Basically, it means that it should be a machine with a 64-bit OS and about 4 to 6 GB of RAM.

- Now Netbeans IDE is ready for the project to be set up. Run it and go to **File | New Project** on the dialog box and select **C/C++ Project with Existing Sources**:



6. Then press **Next** and, on the following dialog box, select the folder with the root of OpenJDK sources:

 You can always obtain your OpenJDK code by typing `hg clone http://hg.openjdk.java.net/jdk6/jdk6 && ./get_source.sh`

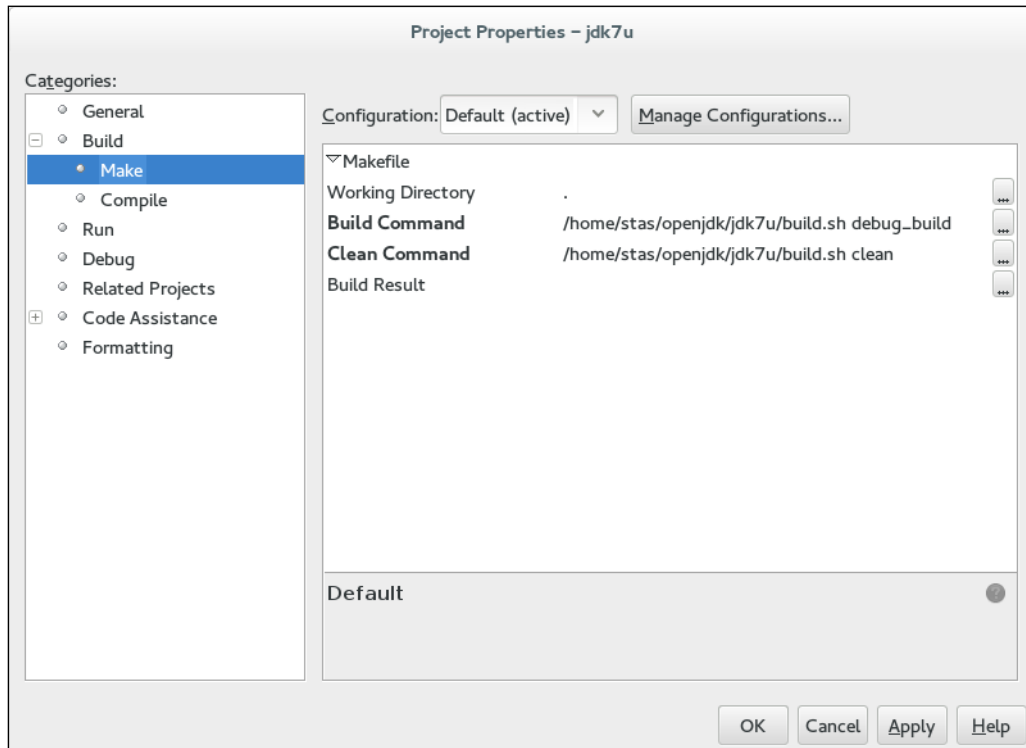
7. Press **Finish** and Netbeans will attempt to clean and build the project. Clean, most probably, will execute without problems, but build will not work because it requires some environment setup, which we will do later on.

After trying to build the project, Netbeans will spend a fair bit of time (minutes) scanning sources and building indexes. This would be a good time to have some coffee.

8. The next step is to configure Netbeans to build the project. As already mentioned, the build script requires some environment setup. The following is a simple bash script which can be used to create an appropriate environment:

```
#!/bin/sh
export LANG=C
export ALT_BOOTDIR=/usr/lib/jvm/java
./jdk/make/jdk_generic_profile.sh
make $*
```

9. In the root folder of the OpenJDK source tree, create a file `build.sh` and save this script in that folder.
10. Then navigate to the **Run | Set project configuration | Customize** menu item and, in the tree on the left-hand side, select **Build | Make**. There you will see the following dialog:



11. As shown in the screenshot, set the **Build Command** and **Clean Command** variables to execute your `./build.sh` with the `debug_build` and `clean` commands, respectively.
12. If the `product` version of OpenJDK is required, then just create another configuration with `product_build` and the parameter for `build.sh`.

See also

Netbeans is not the only available IDE which supports both Java and C++. There are other IDEs which are similarly capable. One example is the Eclipse IDE which is also a powerful multiplatform IDE written in Java, and has similar functionality.

Working with Mercurial forest

Mercurial is a cross-platform version control system. It was designed to work with big projects and large amounts of code, which undoubtedly are present in the OpenJDK project. The OpenJDK official repository is a Mercurial.

The Forest plugin is the one used for various OpenJDK subprojects to merge and coexist. It works with nested Mercurial repositories, which normally are regarded as isolated. The main idea is to propagate changes from the root repository to the nested ones.

The main purpose of it is to allow a developer to work with the code, which is a minor part of a full OpenJDK project repository, without needing to make any changes to the whole repository (change a revision number, for example).

Getting ready

First of all, we will need to install Mercurial itself. On Windows it can be done by going to the official Mercurial site and downloading it from <http://mercurial.selenic.com/wiki/Download>.

For Linux distributions, there are, usually, Mercurial versions in their official repositories.

For example, on Debian and Debian-inherited distributions, Mercurial installs as follows:

```
sudo apt-get install mercurial
```

If you have any problems with a Mercurial installation, refer to the official site or to your Linux distribution resources.

How to do it...

Let's explain it with a simple, non Java-related example. We will assume that a Mercurial instance already exists in the OS. Since Mercurial has a command line tool, we will use a command line for everything.

1. Let's create two repositories:

```
mkdir repo-one
cd repo-one
hg init
echo "hello" > hello.txt
hg ci -m"init one"
cd ..
```

```
mkdir repo-two
cd repo-two
hg init
echo "hello" > hello.txt
hg ci -m"init two"
cd ..
```

2. We will need to locate the .hgrc file:

```
locate hgrc
```

3. Let's copy the forest.py file from <https://bitbucket.org/gxti/hgforest/src>.

4. Then let's edit your .hgrc file:

```
[extensions]
forest = patch/to/forest.py
```

We now have a brand new fclone command in our repository.

1. Let's copy the first repository into the second one:

```
hg clone repo-one repo-two/one
```

The repo-two repository isn't an integral part of repo-one, it only lays inside it.

2. Let's clone a repo-two repository and attach repo-one to it, using the fclone command:

```
hg fclone repo-two wc-two
```

We've just copied repo-two, including repo-two/one.

3. Let's make some changes to `repo-two/hello.txt` and `repo-two/one/hello.txt`:


```
echo some >> repo-two/hello.txt && echo some1 >> repo-two/hello.txt
```
4. We will commit each change using a separate command:


```
cd wc-two/ && ls
hg ci -m"edited hello.txt"
cd one/
hg ci -m"edited hello.txt"
cd ..
```
5. Let's push a result back to `repo-two`:


```
hg fpush
```

We will have two changed files in `repo-one`.
6. Let's push one of them further to `repo-two`:


```
cd ../repo-two
hg fpush
```
7. Now, the changes from `repo-two/one` are propagated to `repo-one`.

How it works...

Mercurial is a relatively simple control system. It is vastly extendable with different plugins, which are configured through the `.hgrc` file.

The Mercurial `forest` plugin propagates changes in nested repositories to the root ones, and synchronizes the parent repository content with a nested one.

Understanding OpenJDK 6 and 7 incremental builds

The process of OpenJDK compilation is very time consuming. It is very boring, especially when one is developing a small part of the whole project, which needs full recompilation for testing purposes. To do it in a simple way and to compile only the necessary parts, there are incremental builds.

Getting ready

We need to download the OpenJDK (6 or 7) source code. You may need `libmotif` installed. Windows users may need to install Cygwin.

How to do it...

We will see how OpenJDK is built incrementally, avoiding adding any nasty bugs.

1. First, let's build OpenJDK for the first time:
`make all`
2. This will take some time, so have a cup of tea.
3. Then, we will build it for the second time:
`make all`
4. You can see from the input that nothing was actually built.
5. Then, let's insignificantly change some source file (for example, `cardTableModRefBS.cpp`).
6. Let's make OpenJDK again, but this time we will `grep` the output:
`make all | grep -i .cpp`
7. We see that, in the output, only two files are actually compiled, `hotspot/src/share/vm/memory/cardTableModRefBS.cpp` and `hotspot/src/share/vm/runtime/vm_version.cpp`.

How it works...

The build program checks the files that were updated and compiles only those which were updated after the last compiler run. However, if any `.hpp` files are modified, the build will be performed in clean mode, for example, no optimization will be performed.

There's more...

Weird things tend to happen while using incremental builds. The probability of such things is increased proportionally with build times.

There are, basically, two ways to perform a clean build:

- ▶ One cleans all files, and compilation from scratch becomes necessary:
`make clean && make all`
- ▶ The second is to specify parameters, which will force the clean mode of the build.

Debugging Java code using NetBeans

Obviously, when someone is writing any code, some debugging is required. It is no surprise that NetBeans, as a high-standard IDE, provides some tools to do that. This recipe will show how to debug the Java code using NetBeans.

Getting ready

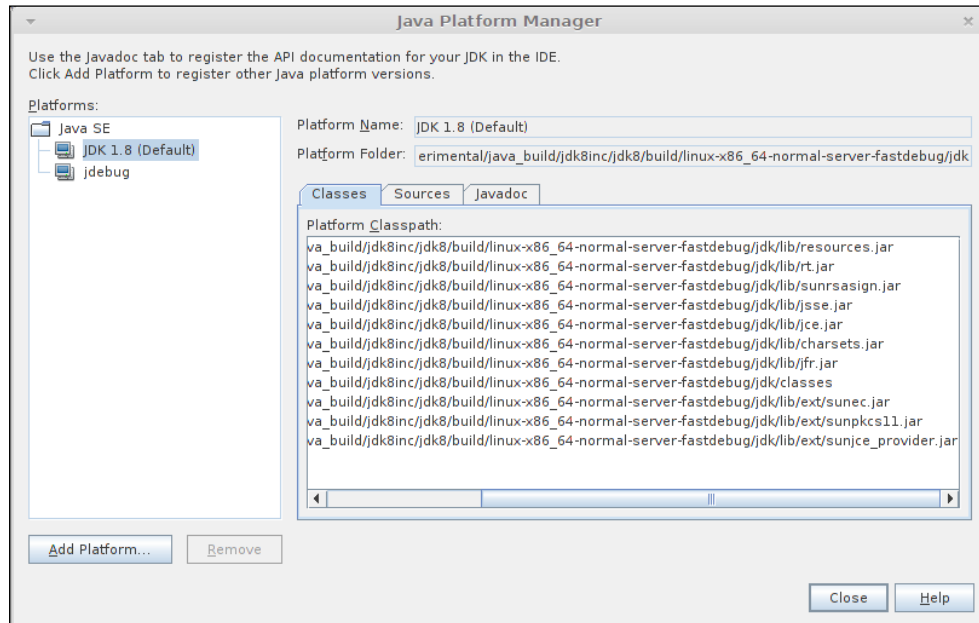
You will need to install NetBeans and set up a development environment, as described previously in the chapter.

How to do it...

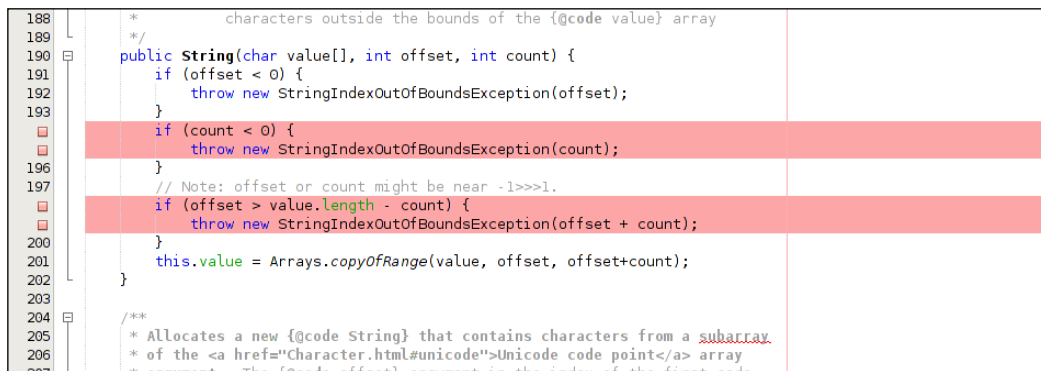
We will use NetBeans to debug our own OpenJDK Java code. We will need to rebuild OpenJDK with debug symbols, and configure NetBeans to make debugging possible:

1. First, let's make our OpenJDK instance with debug symbols:
`bash ./configure --enable-debug`
`make all CONF=linux-x86_64-normal-server-fastdebug`
2. Let's ensure that a debuggable version is built:
`./build/linux-x86_64-normal-server-fastdebug/jdk/bin/java -version`
`openjdk version "1.8.0-internal-fastdebug"`
`OpenJDK Runtime Environment (build 1.8.0-internal-fastdebug-`
`dsmd_2014_03_27_05_34-b00)`
`OpenJDK 64-Bit Server VM (build 25.0-b70-fastdebug, mixed mode)`
3. Now we have a debuggable OpenJDK. Let's set it as the default for NetBeans.
4. Let's open the `etc/netbeans.conf` file in your NetBeans installation path.
5. We will change one line:
`netbeans_jdkhome=<path_to_jdkhome>`
6. After that, we will launch NetBeans and ensure that our JDK is loaded correctly.

7. We will select **Tools | Java Platforms**, and the following screen will appear:



8. Let's try to debug the `java.lang.String` class. We will set our breakpoints to an unavoidable part of this class—to one of the constructors, as shown in the following screenshot:

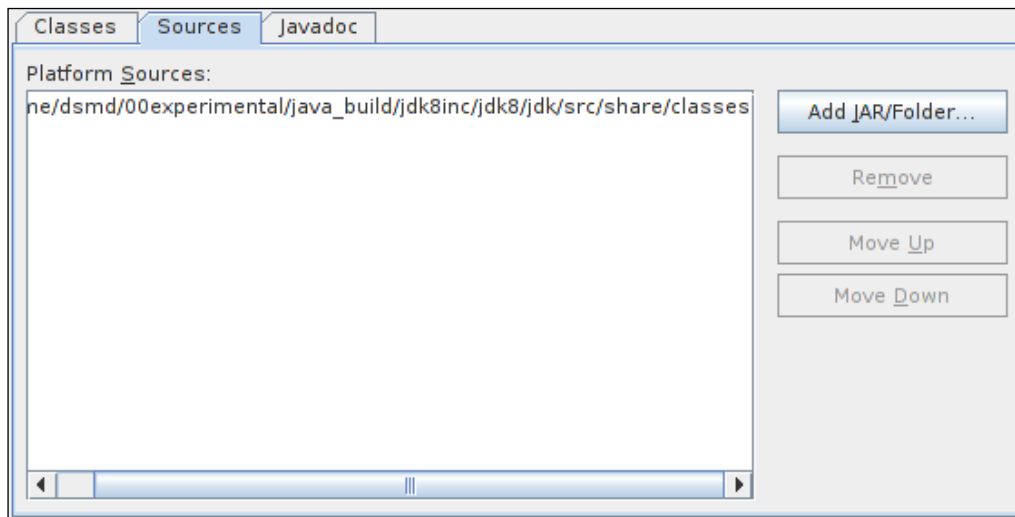


9. This set of breakpoints is sufficient to hook up virtually every Java executable ever launched. But, if we decide to push things forward and attach a debugger, we will get an error message:

```
Not able to submit breakpoint LineBreakpoint String.java
: 138, reason: No source root found for URL 'file:/home/
dsmd/00experimental/java_build/jdk8intr/jdk8/jdk/src/share/
classes/java/lang/String.java'. Verify the setup of project
sources.
```

```
Invalid LineBreakpoint String.java : 138
```

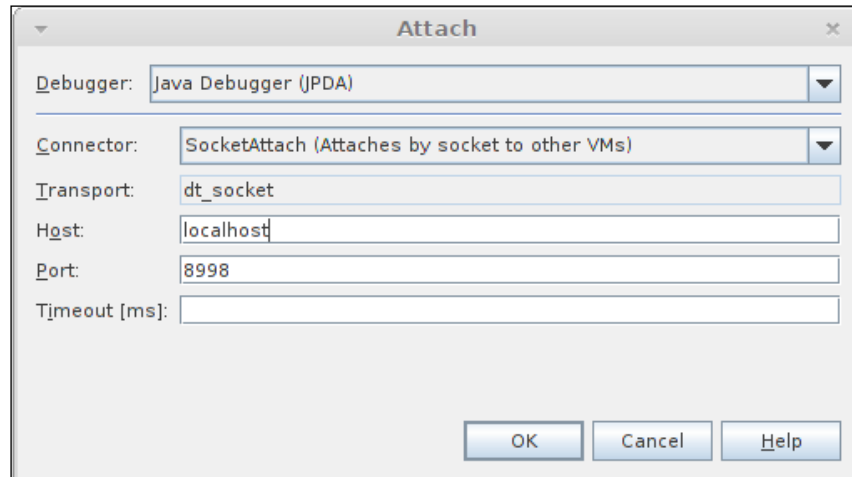
10. To avoid this, we need to specify our Java sources directly to NetBeans. Our project is a C++ project, and it tends to ignore the Java files.
11. The result for the `String` class will be as shown in the following screenshot:



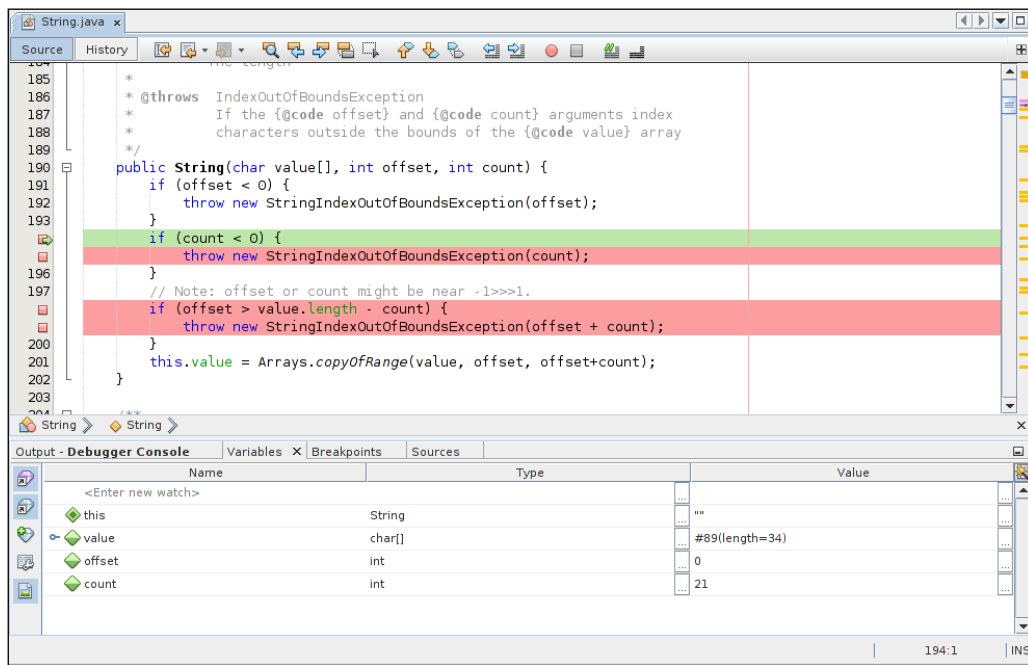
12. Then, just launch some Java executable, that uses strings:

```
build/linux-x86_64-normal-server-fastdebug/jdk/bin/java -Xdebug
-Xrunjdwp:transport=dt_socket,address=8998,server=y -jar /path/to/
jar.jar
```

13. Attach the Java debugger, as shown:



14. Enjoy, you can now see OpenJDK from the inside, in motion:



How it works...

It's just a debugger with a few simple nuances.

There's more...

In some Linux distributions, you can install debug versions of OpenJDK with sources provided in a ZIP file. These sources are automatically picked up by NetBeans.

Debugging C++ code using NetBeans

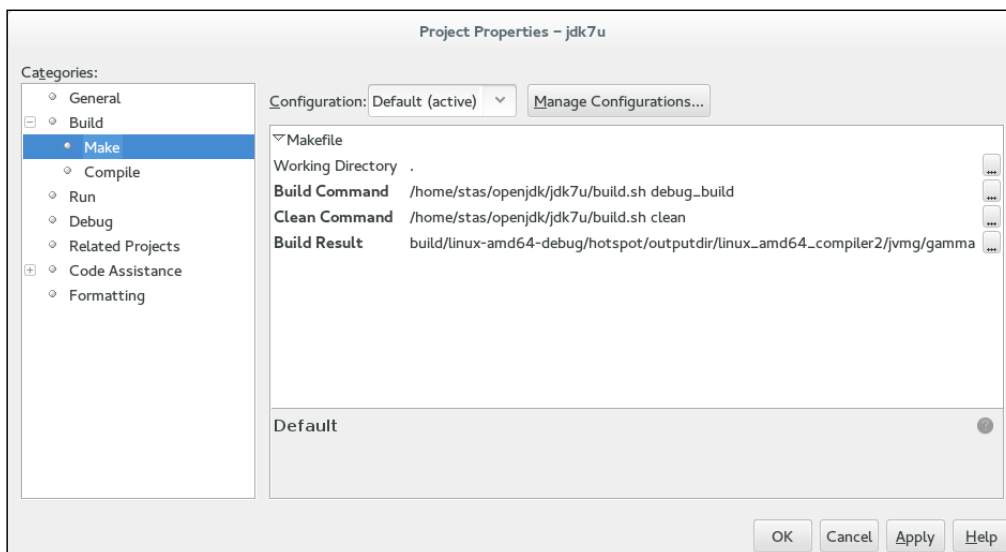
If you plan to make changes to HotSpot or any other C++ part of OpenJDK, then it is certain that you will need to do some step-by-step debugging of the code. This recipe will explain how to set up NetBeans IDE for that purpose.

Getting ready

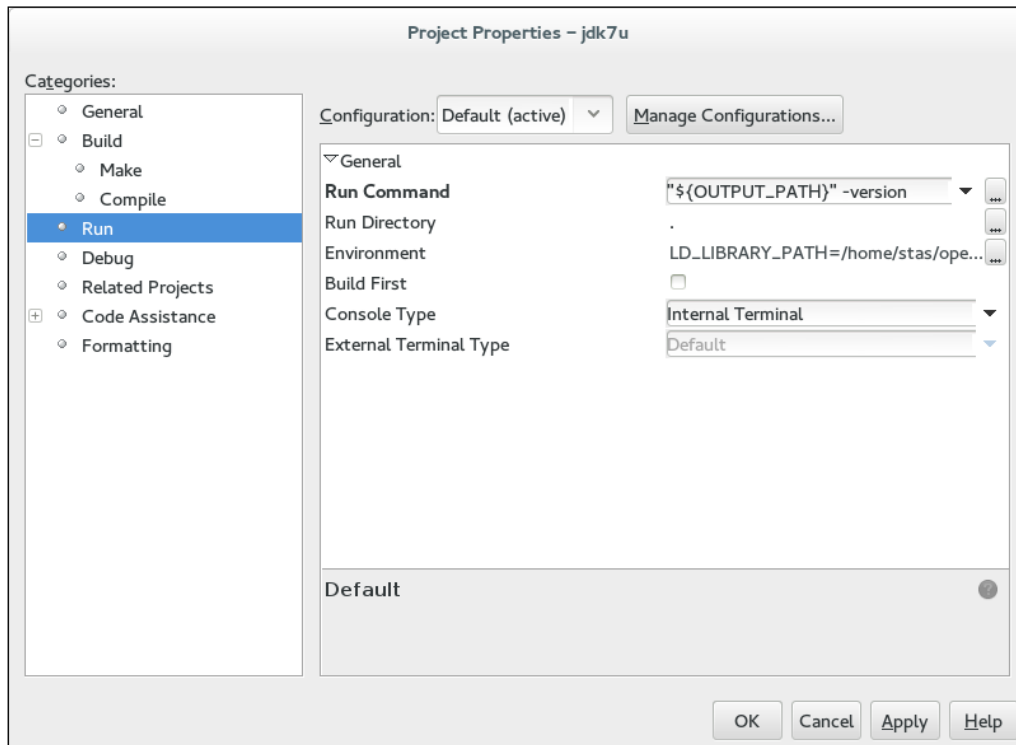
To get started, there are only a few things required—downloaded Open JDK sources and installed NetBeans IDE. It is assumed that the OpenJDK project is already set up and can build sources.

How to do it...

1. The first step is to set up an executable that will run. Go to **Run | Set project configuration / Customize** and then **Build | Make**, and set **build/linux-amd64-debug/hotspot/outputdir/linux_amd64_compiler2/jvmg/gamma** as the build result, as shown in the following screenshot:



2. Then select the **Run** option on the tree on the left had side and set **Run Command** as **"\${OUTPUT_PATH}" -version**:



3. The `-version` flag here is just the simplest thing you can run—get the Java version. You can later change it to whatever you want, for example, to run a Java program.
4. The next step is to set up some environment variables required for Java to run. This can be done by setting them up in the **Environment** section as shown in the following dialog. Set `LD_LIBRARY_PATH` to `build/linux-amd64-debug/hotspot/outputdir/linux_amd64_compiler2/jvms` and `JAVA_HOME` to `build/linux-amd64-debug/j2sdk-server-image`.
5. Now it's all ready to debug. To check that it works, set a breakpoint in `hotspot/src/share/tools/launcher/java.c` somewhere at the beginning of the main function and go to **Debug | Debug Main Project** or use the shortcut `Ctrl + F5`.

How it works...

Careful readers may have noticed that **Debug** used `gamma` JVM launcher, instead of `java`, which is used when you run Java normally. This is to simplify things; `gamma` is the lightweight version of `java`, it doesn't perform checks which are not necessary for debugging purposes.

Using NetBeans to compile HotSpot

When doing HotSpot development, it is very annoying to wait for a complete OpenJDK build to be executed. So, it makes sense to exclude other parts and compile just what we are interested in, that is, the HotSpot part. This recipe will explain how to do that.

Getting ready

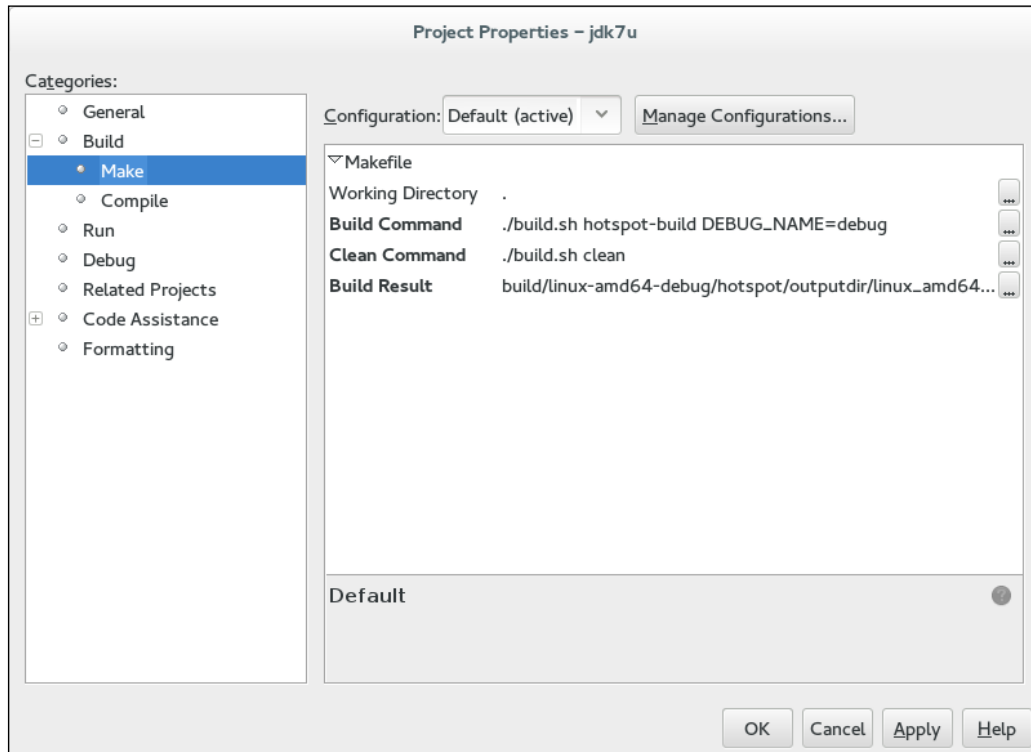
The only prerequisite for this recipe is the availability of source code on the machine, and Netbeans installed, with an OpenJDK project created.

How to do it...

This is a very simple recipe to follow. If you have already completed *Setting up development environment with NetBeans*, the only thing which is required to be done is to change the argument `hotspot_build` and add another argument `DEBUG_NAME=debug`, the whole build command line should look like this:

```
build.sh hotspot-build DEBUG_NAME=debug
```

The **Build/Make** screen of the **Project Properties** dialog in that case will look like this:



How it works...

Luckily, there are targets created in the make configuration which only built the HotSpot bit. These targets can be located in the `./make/hotspot-rules.gmk` file.

The debug command which creates the non-optimized version or the HotSpot, is not the only option for the `DEBUG_NAME` variable. The `fastdebug` command is the other option in which build will create an optimized version with assertions. When `DEBUG_NAME` is not set, the product version of HotSpot is built.

Using HotSpot dev parameters

HotSpot has other options, which may significantly change its behavior. Here we will make use of some of these options, which are used only on the dev versions of OpenJDK.

Getting ready

We will need to compile a dev OpenJDK version, in order to use the dev options.

How to do it...

We will use parameters that are available in the developer version of OpenJDK. In the production builds, they are disabled or set as constant values.

To make use of these parameters, we will run Java as follows:

```
java - -XX:<optionName>
```

Here is a list of some usable dev options:

- ▶ **InlineUnsafeOps:** This option will, if enabled, inline native memory operations from `sun.misc.Unsafe`. It may offer some performance improvements in some cases.
- ▶ **DieOnSafepointTimeout:** This option will kill the process if the safepoint is not reached, but the timeout is exceeded. It is disabled by default.
- ▶ **ZapResourceArea:** This option will zap freed resource / arena space with `0xABABABAB`. It is true in debug mode, but is deselected in production VMs. It may be used for really paranoid security reasons, though it has some performance impact.
- ▶ **ZapJNIHandleArea:** This option will zap freed JNI handle space with `0xFEFEFEFE`. It only has a debug value.
- ▶ **ZapUnusedHeapArea:** This option will zap unused heap space with `0xBAADBABE`. It may be used for security reasons.
- ▶ **Verbose:** This option prints additional debugging information from other modes. It is the main logging option of the dev HotSpot.
- ▶ **UsePrivilegedStack:** This option enables the security JVM functions. It is true by default, but in dev mode, you still have the opportunity to run HotSpot with disabled security.
- ▶ **MemProfiling:** This option writes memory usage profiling to logfile. It is false by default, and can be run for some memory profiling issues.
- ▶ **VerifyParallelOldWithMarkSweep:** This option will use the MarkSweep GC code to verify phases of the parallel old. It may be used for debugging purposes when changing the JVM memory handling mechanism.
- ▶ **ScavengeWithObjectsInToSpace:** This option is really interesting. Java uses two-space GC, where survivor space is concerned, and this options allows scavenges to occur when `to_space` contains objects. Also, when doing so, it clears an unused area, if `ZapUnusedHeapArea` is enabled.

- ▶ `FullGCALot`: This option forces full GC at every *Nth* exit from the runtime system ($N = \text{FullGCALotInterval}$). It may be a very expensive operation, but some developers may build it into JDKs for desktop use. It may be cheaper than to use swap space to absorb endless megabytes from the overgrown heap.
- ▶ `AdaptiveSizePolicyReadyThreshold`: This option is the number of collections before adaptive sizing is started. The default is 5, but it may make sense to make it 1 on desktop systems since the biggest bottleneck of those is the swap space, especially if several Java programs are running simultaneously on one machine.
- ▶ `EagerInitialization`: This option eagerly initializes classes if possible. It is false by default, so maybe it is unsafe to turn it on. But the idea seems good, especially on server machines.
- ▶ `GuaranteedSafepointInterval`: This option guarantees a safepoint (at least) every few milliseconds (0 means none). The default is 1000. It may be used to tweak the stop-the-world state problem. The bigger the option value, the longer these stops will be; and if we make the value too small, we will have too many unnecessary stops.
- ▶ `MaxTrivialSize`: This option is the maximum bytecode size of a trivial method to be inline. It is 6 by default. It is similar to the C++ compiler inline options, but for the bytecode compiler.
- ▶ `MinInliningThreshold`: This option is the minimal invocation count a method needs to have to be inline. It is 250 by default.
- ▶ `SharedOptimizeColdStartPolicy`: This option is the reordering policy for `SharedOptimizeColdStart`. The 0 value favors the classload-time locality, 1 uses a balanced policy, and 2 favors runtime locality.

The default is 2 and it is rarely necessary to change it, but in some cases it will make sense to make it 1 if your application has too many classes that may load after the application starts.

Adding new intrinsic to HotSpot

Intrinsic is a function whose implementation is handled specially by a compiler. Typically, it means that the function call is replaced by automatically generated instructions. This is very similar to the inline functions, but the compiler knows more about intrinsics as they are part of the compiler itself, so it can use them more wisely.

Intrinsics are usually more performance-oriented than native functions because there is no JNI overhead.

Getting ready

To get started, all we need is an installed NetBeans IDE for code editing and OpenJDK sources. The user should be able to read C++ code and a little knowledge of assembly would be beneficial.

It would be worth checking that SSE4.2 (an extended instruction set with six new commands, mostly for character searching and comparison) is supported by the CPU (the `CRC32` command used in our example is from that command set). It was introduced with Core i7 Intel chips back in 2009, so if you are using an Intel CPU, it should be present. The first time it was introduced for AMD was as Bulldozer chips back in 2011, so you should have relatively recent chips to support it. If your CPU is not compatible with that command, do not worry. The recipe is applicable to any intrinsics you may want to introduce; there is no difference, apart from the actual implementation of the code you want to intrinsify.

How to do it...

Adding new intrinsics is not a simple process. Follow these instructions carefully. Ensure you compile the code after every step; doing so may save some time.

The instruction which we are going to intrinsify is the CRC32 calculation, which is implemented by `java.util.zip.CRC32`.

To start, let's make a small amendment to the Java class that is responsible for the CRC32 calculation. We are going to add a method which will be intrinsified by HotSpot. Open the `jdk/src/share/classes/java/util/zip/CRC32.java` file and add a new method `doUpdateBytes`:

```
private static int doUpdateBytes(int crc, byte[] b, int off, int len)
{
    return updateBytes(crc, b, off, len);
}
```

That implementation just makes a call to the currently used `updateBytes` native method. That's the only change in Java. The rest is going to be the C++ internals of the HotSpot.

Open the `hotspot/src/share/vm/precompiled/precompiled.hpp` file and add the following line into it:

```
#include "smmintrin.h"
```

The `smmintrin.h` file contains GCC intrinsics, which are going to be used by our implementation of the CRC32 function.

Then, as we are using SSE4.2 instructions, we need to inform the compiler. To do so, open the `hotspot/make/linux/makefiles/gcc.make` file (assuming you are making the build on Linux), and locate the line consisting of `CFLAGS += -fno-rtti`. Just after that line, add the `-msse4.2` flag, so it will look like this:

```
CFLAGS += -fno-rtti # locate that line
CFLAGS += -msse4.2  # add this new line here
```

Now we are ready to implement our CRC32 function in C++. In the `hotspot/src/cpu/x86/vm/` folder, create the `CRC32Calc` class and the `static_calcCrc32` static method. Here is the `CRC32Calc.hpp` file with the class declaration:

```
#ifndef CRC32CALC_HPP
#define CRC32CALC_HPP

class CRC32Calc {
public:
    CRC32Calc() {}
    virtual ~CRC32Calc() {}

    static int static_calcCrc32(int crc, const char* data, int
dataOffset, int dataLen);
};

#endif /* CRC32CALC_HPP */
```

The `CRC32Calc.cpp` file with its implementation is as shown:

```
#include "CRC32Calc.hpp"
#include "precompiled.hpp"
int CRC32Calc::static_calcCrc32(int crc, const char* data, int
dataOffset, int dataLen) {
    const int dataSize = dataLen - dataOffset;
    int result = crc;
    int uints32 = (int)(dataSize / sizeof(int));
    int units8 = dataSize % sizeof(int);

    const int* pUint32 = (const int*)data;
    while (uints32--) {
        result = ::_mm_crc32_u32(result, *pUint32);
        pUint32++;
    }

    const char* pUnit8 = (const char*)pUint32;
    while (units8--) {
```

```

        result = ::_mm_crc32_u8(result, *pUnit8);
        pUnit8++;
    }

    return result;
}

```

The following instructs HotSpot how to intrinsify our method.

Locate `hotspot/src/share/vm/classfile/vmSymbols.hpp`. That is the file which contains the declaration of all intrinsics and add the following definition to it:

```

do_class(java_util_zip_crc32,      "java/util/zip/CRC32")
\
do_intrinsic(_crc32_doUpdateBytes, java_util_zip_crc32, doUpdateBytes_
name, int_byteArray_int_int_signature, F_R) \
do_name(      doUpdateBytes_name,          \
"doUpdateBytes")
do_signature(int_byteArray_int_int_signature,      " (I [BII) I"
)
\

```

This is the declaration of intrinsics which maps the Java method with the code which will replace it in the runtime. Be careful when adding it. It is based on macros, which means, if there is a typo or any other mistake, it will be very hard to figure out where the problem is.

The next step is to define which code we are going to generate for the intrinsic. We are not going to be very smart here, as this is just an exercise to see how the functionality works. So all our assembler is going to do is generate a call to the C function. Add the following into `hotspot/src/cpu/x86/vm/stubGenerator_x86_64.cpp` and `hotspot/src/cpu/x86/vm/stubGenerator_x86_32.cpp`:

```
#include "CRC32Calc.hpp"
```

Now it's a bit tricky and requires some low-level code. We are going to tell HotSpot how to generate the assembly for our method. To do so, add the `generator` method into the `StubGenerator` class which is declared in both `hotspot/src/cpu/x86/vm/stubGenerator_x86_64.cpp` and `hotspot/src/cpu/x86/vm/stubGenerator_x86_32.cpp` for `x86_64` and `x86` architectures respectively. The code for the method is as follows:

```

// Arguments:
//
// Inputs:
//   c_rarg0   - input crc
//   c_rarg1   - byte array with data for calculation
//   c_rarg2   - offset in the input array

```

```
//  c_rarg3    - number of data bytes after offset
//
//  Output:
//    eax - result crc
address generate_crc32_doUpdateBytes() {
    __ align(CodeEntryAlignment);
    StubCodeMark mark(this, "StubRoutines", "crc32_doUpdateBytes");
    address start = __ pc();

    __ enter(); // required for proper stackwalking of RuntimeStub
frame
    __ pusha();
    // no need to put params in regr - they are already there
    // after this call rax should already have required return value
    __ call_VM_leaf(CAST_FROM_FN_PTR(address, CRC32Calc::static_
calcCrc32), 4);
    __ popa();

    __ leave(); // required for proper stackwalking of RuntimeStub
frame
    return start;
}
```

Now we need a variable which will contain the address of the generated method. To do so, add the following static member declaration to `hotspot/src/share/vm/runtime/stubRoutines.hpp`:

```
static address _crc32_doUpdateBytes;
```

To the same file, add the following method, which just returns the value of the declared variable:

```
static address crc32_doUpdateBytes() { return _crc32_doUpdateBytes; }
```

Then, in `hotspot/src/share/vm/runtime/stubRoutines.cpp`, assign a default value to `_crc32_doUpdateBytes`:

```
address StubRoutines::_crc32_doUpdateBytes = NULL;
```

Then, in both `hotspot/src/cpu/x86/vm/stubGenerator_x86_64.cpp` and `hotspot/src/cpu/x86/vm/stubGenerator_x86_32.cpp`, locate the `generate_all` method and assign the following value to the variable `_crc32_doUpdateBytes`:

```
StubRoutines::_crc32_doUpdateBytes = generate_crc32_doUpdateBytes();
```

The next step is to add the method which creates a descriptor. The descriptor is the definition of our function—how many arguments it takes, which types of arguments it accepts, and so on. The first step is to add the method declaration into the `OptoRuntime` class in the `hotspot/src/share/vm/opto/runtime.hpp` file:

```
static const TypeFunc* crc32_Type();
```

This will be the function which creates the type information for our method call—it describes the arguments and returns the parameters. After implementation it creates an array of types of input parameters and the type of the return value. Place it in the `hotspot/src/share/vm/opto/runtime.cpp` file:

```
const TypeFunc* OptoRuntime::crc32_Type() {
    // create input type (domain): int, pointer, int, int
    int num_args      = 4;
    int argcnt = num_args;
    const Type** fields = TypeTuple::fields(argcnt);
    int argp = TypeFunc::Parms;
    fields[argp++] = TypeInt::INT;      // crc
    fields[argp++] = TypePtr::NOTNULL;  // data
    fields[argp++] = TypeInt::INT;      // offset
    fields[argp++] = TypeInt::INT;      // len
    const TypeTuple* domain = TypeTuple::make(TypeFunc::Parms+argcnt,
        fields);

    // create return value
    fields = TypeTuple::fields(1);
    fields[TypeFunc::Parms+0] = TypeInt::INT;

    const TypeTuple* range = TypeTuple::make(TypeFunc::Parms+1, fields);

    return TypeFunc::make(domain, range);
}
```

Now we will implement the method that will inline the code in the runtime. In the `hotspot/src/share/vm/opto/library_call.cpp` file, locate the definition of the `LibraryCallKit` class, and add the following method declaration:

```
bool inline_crc32();
```

Also, in the same file, add the implementation:

```
bool LibraryCallKit::inline_crc32() {
    address stubAddr = StubRoutines::crc32_doUpdateBytes();
    const char *stubName = "crc32_doUpdateBytes";
```

```
Node* inputCrc = argument(0);
Node* in_data  = argument(1);
Node* offset   = argument(2);
Node* len      = argument(3);

// Call the stub.
make_runtime_call(RC_LEAF|RC_NO_FP, OptoRuntime::crc32_Type(),
                  stubAddr, stubName, TypePtr::BOTTOM,
                  inputCrc, in_data, offset, len);

return true;
}
```

Finally tell HotSpot that we indeed want to intrinsify our method call and make a call to the inlining method `inline_crc32`.

To tell HotSpot that we want to intrinsify the method, `Compile::make_vm_intrinsic` method in the file `hotspot/src/share/vm/opto/library_call.cpp` has to return a non-null pointer to `CallGenerator`. To do so, add the following line into the `switch(id)` switch statement in that method:

```
case vmIntrinsics::_crc32_doUpdateBytes:
    break;
```

It is not strictly required to have that case and break, the default works just well; but it makes it more explicit that we are using intrinsics for the CRC32 calculation method.

Then, to make a call to the inlining method, in the same `hotspot/src/share/vm/opto/library_call.cpp` file, locate `LibraryCallKit::try_to_inline`, find `switch(intrinsic_id())`, and add the following line of code:

```
case vmIntrinsics::_crc32_doUpdateBytes: return inline_crc32();
new line.
```

How it works...

To check whether the method was intrinsified, use the `-XX:+PrintCompilation` and `-XX:+PrintInlining` Java arguments. To see what the intrinsics are compiled into, use `-XX:+PrintAssembly` (this should be prepended by `-XX:+UnlockDiagnosticsVMOptions` when running on the product build).

There's more...

To see if SSE4.2 is supported, just compile and run the following code:

```
// This is Linux version
#include <cpuid.h>
#include <stdio.h>
void main () {
    unsigned int eax, ebx, ecx, edx;
    __get_cpuid(1, &eax, &ebx, &ecx, &edx);
    if (ecx & bit_SSE4_2)
        printf ("SSE4.2 is supported\n");
    return;
}

// And this is the version for windows
#include <intrin.h>
int _tmain(int argc, _TCHAR* argv[])
{
    int cpuInfo[4] = { -1 };
    __cpuid(cpuInfo, 1);
    bool bSSE42Extensions = (cpuInfo[2] & 0x100000) || false;
    if (bSSE42Extensions) {
        printf("SSE4.2 is supported\n");
    }
    return 0;
}
```

There are lots of intrinsic methods. See `library_call.cpp` and `vmSymbols.hpp`:

- ▶ `Object.getClass` gives one or two instructions.
- ▶ `Class.isInstance` and `Class.isAssignableFrom` are as cheap as instances of bytecodes when the operands are constants, and otherwise no more expensive than astore type checks.
- ▶ Most single-bit class queries are cheap and even constant-foldable.
- ▶ Reflective array creation is about as cheap as the `newarray` or `anewarray` instructions.
- ▶ `Object.clone` is cheap and shares code with `Arrays.copyOf` (after Java6).

Java is not the only language which uses intrinsics they are also widely used in C++ for SSE operations.

Interestingly, `_mm_crc32_u32` and `_mm_crc32_u8` are intrinsics themselves, known by the GCC or MS compiler, which are directly replaced by assembly instructions in the compiled code.

Building VisualVM from the source code

VisualVM is an open source project which is not a part of OpenJDK. It is a powerful tool which is helpful to anyone who uses applications based on JDK. It allows us to monitor parameters of the system, browse heap dumps, create thread dumps, and so on. As the tool is open source, it is possible to get the source code and customize it as required, or simply just to see how it works. This recipe will go through the steps which are required to download the source code and build VisualVM from it.

Getting ready

This recipe requires a machine with Subversion and Ant installed. Also, as VisualVM is a graphical application, a graphical environment is required to run it. It is possible to perform a build without launching the application.

How to do it...

The first step is to get the source code:

1. Create a folder for the sources, for example, `/home/user/visualvm`.
2. Go to the newly created folder and, assuming you need sources from the `trunk`, run the following command:

```
svn checkout https://svn.java.net/svn/visualvm~svn/trunk
```
3. This will create the `trunk` folder with sources in the current directory.

To start, we need to download the NetBeans Platform binaries. The version needed depends on the version of VisualVM we are going to build. In this example, we will use `trunk`, the current development version which requires NetBeans Platform v.8; but as that may change, it is recommended to consult the page for the appropriate version using the link <http://visualvm.java.net/build.html>. These binaries are available directly from the VisualVM website, not from the NetBeans website. For this example, the URL is https://java.net/projects/visualvm/downloads/download/dev/nb80_visualvm_27062014.zip. When the file is downloaded, unpack it into the `trunk/visualvm` folder, as shown:

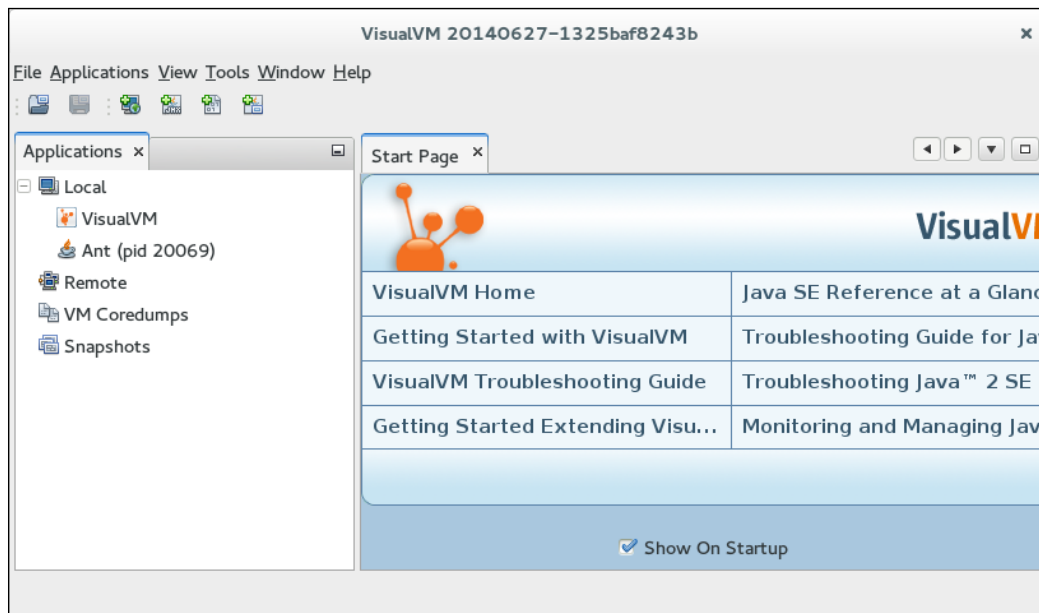
1. Now, execute Ant to run the build.

```
ant build-zip
```
2. When the build is complete, we should see something similar to the following output in the command prompt:

```
BUILD SUCCESSFUL
Total time: 34 seconds
```

This indicates that the build was successful. This step is not required if we just need to run VisualVM, as Ant will also run the build target; but if there is no need to run, and just build is required, this step can be useful.

3. To run VisualVM, run the following command:
ant run
4. If the application is not built yet, then Ant will build it first and then run it. As VisualVM is a GUI application, we will see the following screen:



This is the landing screen of VisualVM. The fact that we can see it means that the application is built and works properly. The zipped archive file with the distribution can be found in the `visualvm/dist` folder.

See also

- More information about VisualVM build is available on the home page <http://visualvm.java.net/build.html>. Each version has a different page, as build instructions vary from version to version. For example, each build may require a slightly different version of the Netbeans platform.

Creating a plugin for VisualVM

VisualVM is just an application which has a predefined and limited set of features. It is a framework, which means that it is possible to extend it. The way in which VisualVM provides for extension is via the plugin API which allows us to create new plugins which are then available through the application. Such plugins can do various things, but are mostly used to provide new ways of monitoring or controlling the JVM applications.

Getting ready

The current (at the time of writing) trunk version of VisualVM requires the NetBeans platform and IDE v.8. So we need to ensure that the current version of the platform is available on the machine. If there is any doubt, check the page with VisualVM trunk build instructions at <http://visualvm.java.net/build/build.html>.

How to do it...

Let's start with what we are going to monitor. It seems like the simplest thing we can do is to build a component which will tick the data that we can read. For example, look at the following class:

```
package org.openjdk.cookbook;

import javax.management.MBeanServer;
import javax.management.ObjectName;
import java.lang.management.ManagementFactory;

public class SleepProbe implements SleepProbeMBean {
    private volatile long lastSleepSampleMs = 100;

    public static void main(String[] args) throws Exception {
        MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
        ObjectName name = new ObjectName("org.openjdk.
cookbook:type=SleepProbe");
        SleepProbe mbean = new SleepProbe();
        mbean.start();
        mbs.registerMBean(mbean, name);
        System.out.println("Started MBean");
        Thread.sleep(Long.MAX_VALUE);
    }

    @Override
```

```

    public long getActualSleepTime() {
        return lastSleepSample;
    }

    public void start() {
        new Thread(new Runnable() {
            @Override
            public void run() {
                while ( !Thread.currentThread().isInterrupted() ) {
                    try {
                        final long start = System.nanoTime();
                        Thread.sleep(100);
                        final long end = System.nanoTime();
                        lastSleepSampleMs = (long)((double)(end-
start))/1000000;
                    } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                        break;
                    }
                }
            }
        }).start();
    }
}

```

This code sleeps for 100 ms and measures how long it actually slept. The value is not going to be exact, but will be about 100. It publishes the last measurement of the sleep time via the `lastSleepSample` variable which is available via the `SleepProbeMBean` interface:

```

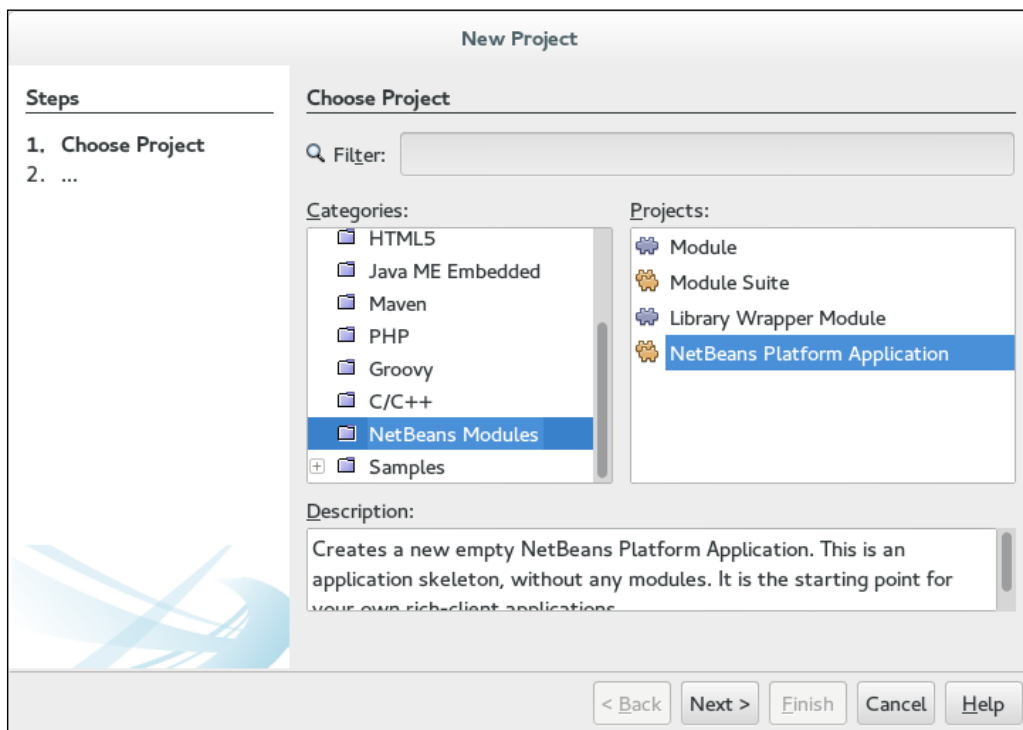
package org.openjdk.cookbook;

public interface SleepProbeMBean {
    public long getActualSleepTime();
}

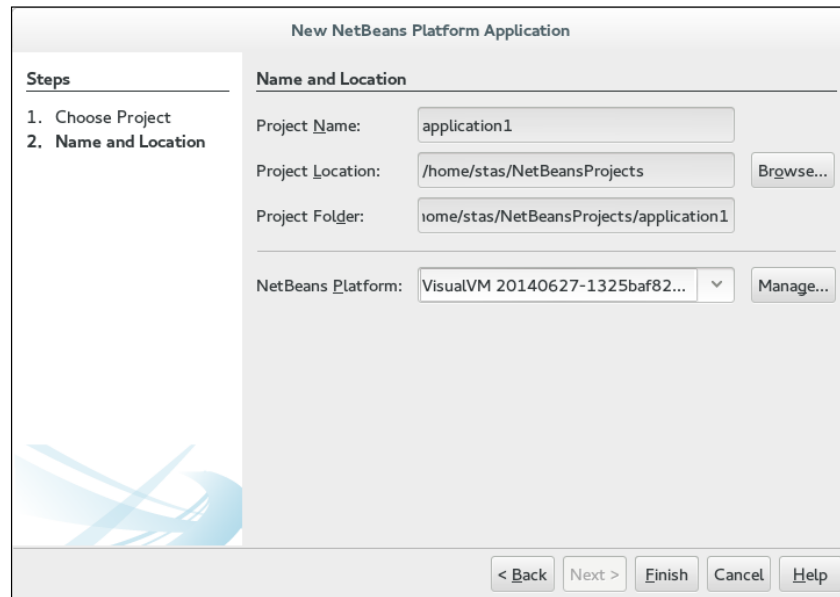
```

This class and interface should be put into a separate project, so you can run them independently of the VirtualVM plugin project:

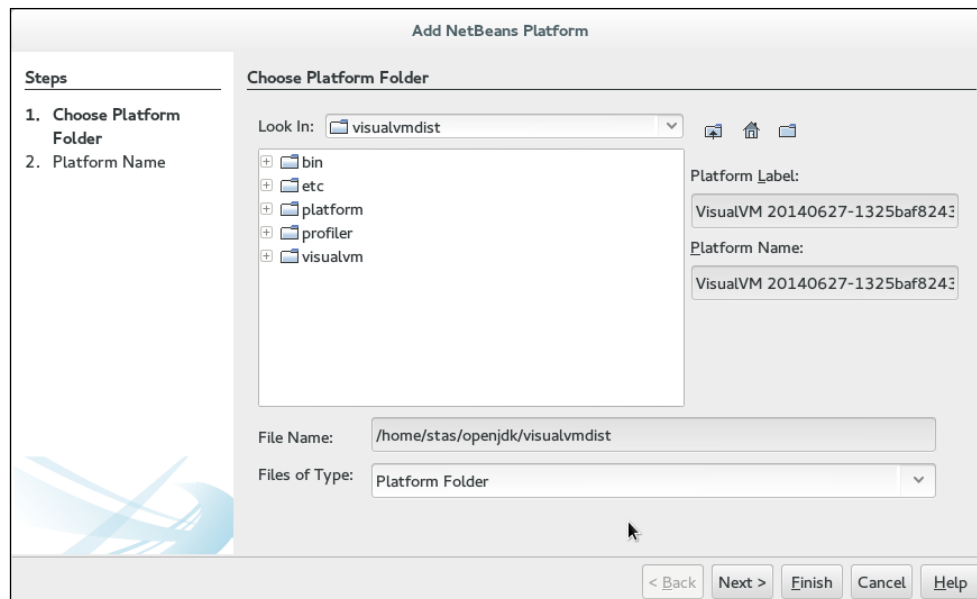
1. To start, we need to create a plugin project in IDE. Launch the IDE, go to **File | New Project** and select **NetBeans Platform Application** from the project type:



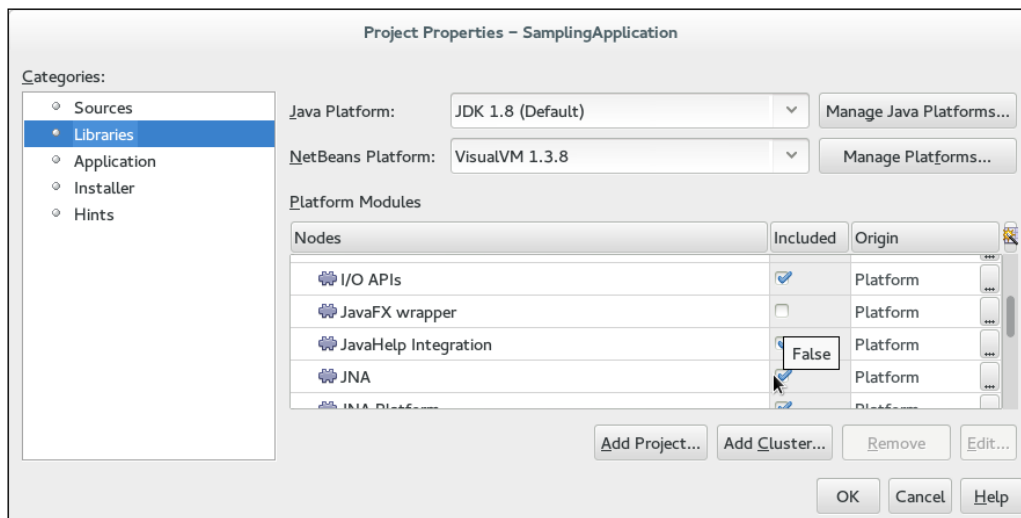
- On the next screen, select **VisualVM** as the **NetBeans Platform** (if it is not available, see further instructions), the project name, and the location, as shown:



- If **VisualVM** is not available in the list of platforms, then click on the Manage icon and, in the displayed dialog, add a new platform by pointing it to the folder with the VisualVM distribution, as shown in the following screenshot:

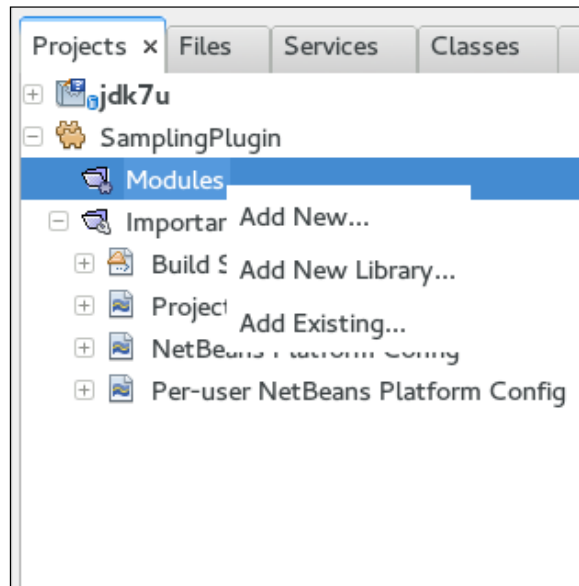


4. In this example, the distribution was built from source (see the *Building VisualVM from the source code* recipe). Press **Next** and then **Finish**.
5. Now just complete the wizard and you will have a new empty project with some properties and build script.
6. Note that there is a known bug in some versions of NetBeans (https://netbeans.org/bugzilla/show_bug.cgi?id=242564), which is causing problem with dependencies and which will not allow us to add the required dependencies later. To work around this, right click on the project and then click on **Properties**. On the **Project Properties** dialog, select **Libraries**:



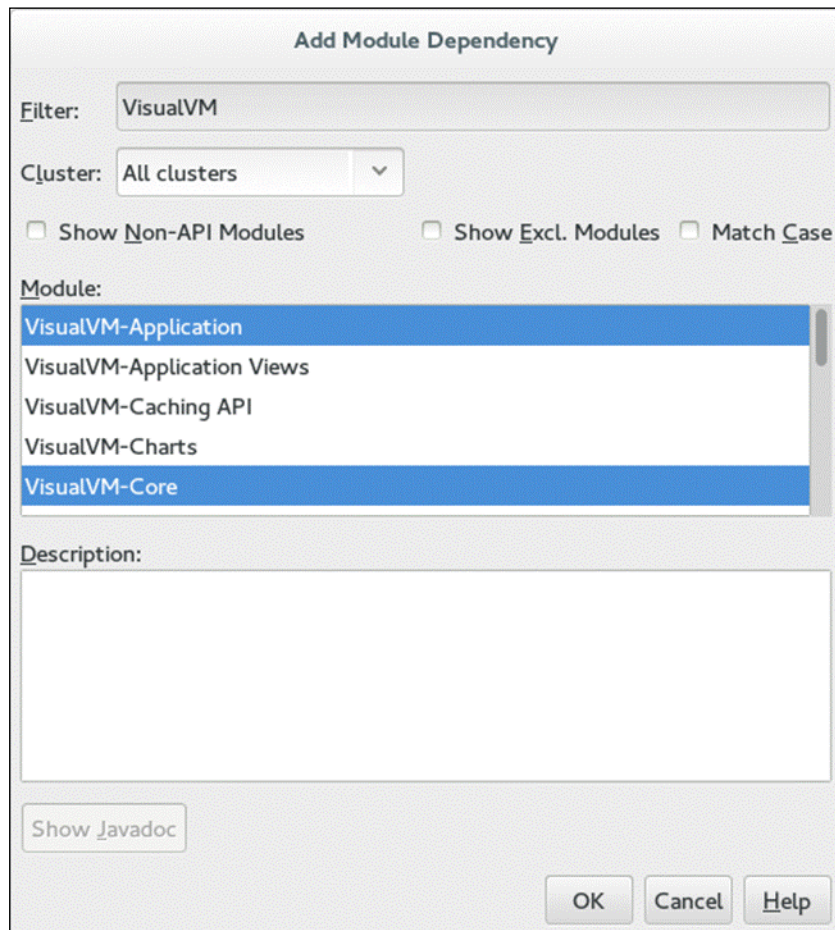
7. Uncheck **JavaFX wrapper** under the platform component. Ensure that all the other checkboxes are checked, including **profiles** and **visualvm** nodes.

8. Now we need to create a module for our plugin. Right click on the **Modules** item in the project tree and select **Add New...**:

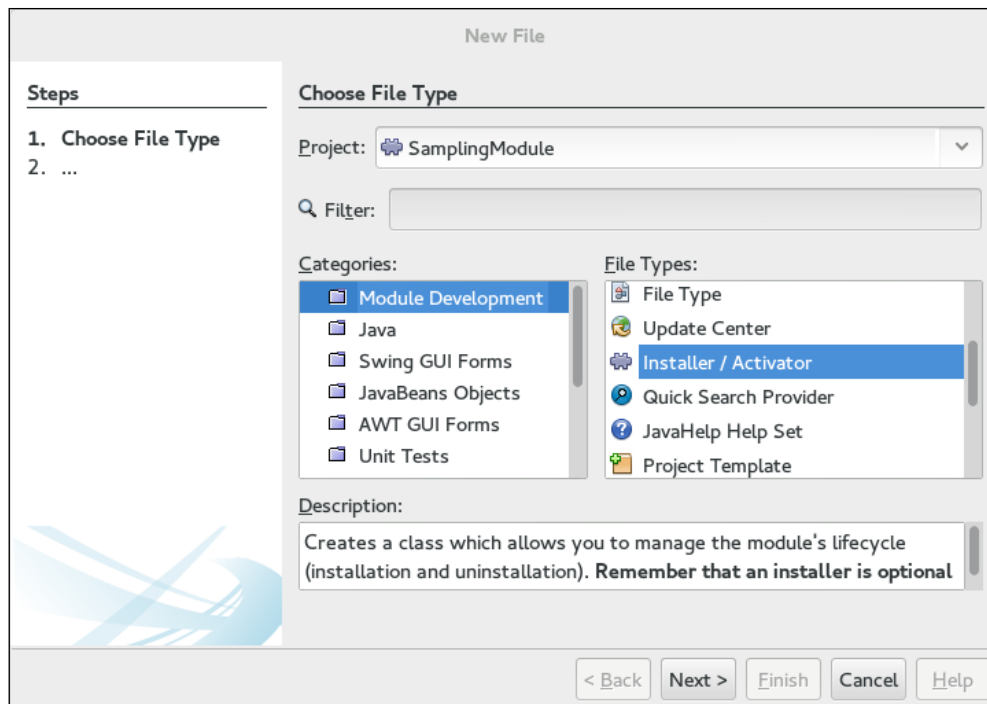


9. Name it as `SamplingModule` and press **Next**. On the next screen, put `net.cookbook.openjdk` as Code Name Base and press on **Finish**. This will create an empty module where we will need to add some components.

10. The next step is to add the dependencies to the module. Right click on the module and select **Properties**, then go to **Libraries | Module Dependencies**, and click on **Add Dependency**. On the **Add Module Dependency** dialog, put `VisualVM` into the **Filter** field as shown:



11. Select **VisualVM-Application**, **VisualVM-Core**, and **VisualVM-Tools** (not visible on the screenshot), then click on **OK**.
12. The next step is to add the installer and some source code. To do so, right click on the newly created module and go to **New | Other**. This will show the dialog with the option to select the file type. Click on **Installer/Activator** and click on **Next**, as shown:



13. Then just complete the wizard by clicking on **Finish**. This will create a class with the name `Installer` in the package `net.cookbook.openjdk`. For now, leave that class as it is, we will update it later in the recipe.
14. The next step is to create a component which will draw a graph for us. To do so, we will create a simple panel which will refresh itself every half a second with a new sample from the sampler `MBean`. In the `net.cookbook.openjdk` package, create a new class and name it as `SamplingGraphPanel`:

```
package net.cookbook.openjdk;

import com.sun.tools.visualvm.application.Application;
import com.sun.tools.visualvm.tools.jmx.*;
import java.awt.*;
import java.util.LinkedList;
import javax.management.*;
import javax.swing.JPanel;
import org.openide.util.Exceptions;

public class SamplingGraphPanel extends JPanel implements Runnable
{
    private static final int MAX_DATA_POINTS = 20;
```

```
private static final int MAX_VALUE = 110;
private static final int GAP = 30;

private final LinkedList<Long> samples = new
LinkedList<Long>();
private final Application application;
private Thread refreshThread;

public SamplingGraphPanel(Application application) {
    this.application = application;
    this.setBackground(Color.black);
}

@Override
public void paintComponent(Graphics gr) {
    super.paintComponent(gr);

    Graphics2D g2 = (Graphics2D)gr;
    g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);

    final double xScale = ((double) getWidth()-2*GAP)/
(samples.size()-1);
    final double yScale = ((double) getHeight()-2*GAP)/(MAX_
VALUE-1);

    Stroke oldStroke = g2.getStroke();
    g2.setColor(Color.green);
    g2.setStroke(new BasicStroke(3f));
    for (int i = 0; i < samples.size()-1; ++i) {
        final int x1 = (int) (i * xScale + GAP);
        final int y1 = (int) ((MAX_VALUE-samples.
get(i))*yScale+GAP);
        final int x2 = (int) ((i+1) * xScale + GAP);
        final int y2 = (int) ((MAX_VALUE - samples.get(i+1)) *
yScale + GAP);
        g2.drawLine(x1, y1, x2, y2);
    }
}

public void start() {
    refreshThread = new Thread(this);
    refreshThread.start();
}
```

```
}

public void stop() {
    if ( refreshThread != null ) {
        refreshThread.interrupt();
        refreshThread = null;
    }
}

@Override
public void run() {
    JmxModel jmx = JmxModelFactory.
getJmxModelFor(application);
    MBeanServerConnection mbsc = null;
    if (jmx != null && jmx.getConnectionState() == JmxModel.
ConnectionState.CONNECTED) {
        mbsc = jmx.getMBeanServerConnection();
    }

    try {
        while ( mbsc != null && !Thread.currentThread().
isInterrupted() ) {
            if ( samples.size() == MAX_DATA_POINTS ) {
                samples.remove();
            }
            Long val = (Long)mbsc.getAttribute(new
ObjectName("org.openjdk.cookbook:type=SleepProbe"),
"ActualSleepTime");
            samples.add(val);
            repaint();
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) { break; }
        }
    } catch (Exception e) {
        Exceptions.printStackTrace(e);
    }
}
}
```

15. This class will read a value from the MBean implements on the first step of this receipt every 500ms, and add the value to the list with samples. Then it refreshes the graph, which gets repainted. Fundamentally, that code is just a Java Swing code, which can be run in any application. The only VisualVM-specific bit here is some helper classes used to get MBean from the Application object.
16. Now, in SamplingModule, create a class which will be responsible for showing the data. Name it SamplingView and put it into the `net.cookbook.openjdk` package, as shown below:

```
package net.cookbook.openjdk;

import com.sun.tools.visualvm.application.Application;
import com.sun.tools.visualvm.core.ui.DataSourceView;
import com.sun.tools.visualvm.core.ui.components.
DataViewComponent;
import javax.swing.*;
import org.openide.util.Utilities;

public class SamplingView extends DataSourceView {
    private DataViewComponent dvc;
    private SamplingGraphPanel panel;
    public static final String IMAGE_PATH = "net/cookbook/openjdk/
icon.png";

    public SamplingView(Application application) {
        super(application, "Sampling Application", new
ImageIcon(Utilities.loadImage(IMAGE_PATH, true)).getImage(), 60,
false);
    }

    protected DataViewComponent createComponent() {
        //Data area for master view:
        JEditorPane generalDataArea = new JEditorPane();
        generalDataArea.setBorder(BorderFactory.
createEmptyBorder(14, 8, 14, 8));

        panel = new SamplingGraphPanel(SamplingProvider.
getSleepProbeInstance((Application)getDataSource()));
        DataViewComponent.MasterView masterView = new DataViewComponent.
MasterView("Sampling Overview", null, generalDataArea);
        DataViewComponent.MasterViewConfiguration
masterConfiguration = new DataViewComponent.
MasterViewConfiguration(false);
        dvc = new DataViewComponent(masterView,
masterConfiguration);
        //Add detail views to the component:
```

```

        dvc.addDetailsView(new DataViewComponent.
DetailsView("Sampling Graph", null, 10, panel, null),
DataViewComponent.TOP_LEFT);

        return dvc;
    }

    @Override
    protected void removed() {
        super.removed();
        panel.stop();
    }
}

```

17. It is important that the file referenced by `IMAGE_PATH` actually exists, otherwise the plugin will not start and will fail with an exception. The simplest way you can do this is to download any of available free icons from the Internet, for example at <https://www.iconfinder.com/icons/131715/download/png/32> and put it into the same package folder as the `SamplingView` class.
18. The next step is to create a provider which will create the view instance and identify that the application we are connected to is supported by the plugin. In the `net.cookbook.openjdk` package, create a class with the name `SamplingProvider` and with the following implementation:

```

package net.cookbook.openjdk;

import com.sun.tools.visualvm.application.Application;
import com.sun.tools.visualvm.core.ui.*;
import com.sun.tools.visualvm.tools.jmx.*;
import javax.management.*;
import org.openide.util.Exceptions;

public class SamplingProvider extends DataSourceViewProvider<Application> {
    private static DataSourceViewProvider instance = new
SamplingProvider();
    @Override
    public boolean supportsViewFor(Application application) {
        boolean result = false;
        JmxModel jmx = JmxModelFactory.
getJmxModelFor(application);
        if (jmx != null && jmx.getConnectionState() == JmxModel.
ConnectionState.CONNECTED) {
            MBeanServerConnection mbsc = jmx.
getMBeanServerConnection();
            if (mbsc != null) {

```



```
        try {
            mbsc.getObjectInstance(new ObjectName("org.
openjdk.cookbook:type=SleepProbe"));
            result = true; // no exception - bean found
        } catch (InstanceNotFoundException e) {
            // bean not found, ignore
        } catch (Exception e1) {
            Exceptions.printStackTrace(e1);
        }
    }
}
return result;
}

@Override
protected DataSourceView createView(Application application) {
    return new SamplingView(application);
}

static void initialize() {
    DataSourceViewsManager.sharedInstance().
addViewProvider(instance, Application.class);
}

static void unregister() {
    DataSourceViewsManager.sharedInstance()
        .removeViewProvider(instance);
}

public static Object getSleepProbeInstance(Application
application) {
    ObjectInstance instance = null;
    JmxModel jmx = JmxModelFactory.
getJmxModelFor(application);
    if (jmx != null && jmx.getConnectionState() == JmxModel.
ConnectionState.CONNECTED) {
        MBeanServerConnection mbsc = jmx.
getMBeanServerConnection();
        if (mbsc != null) {
            try {
                instance = mbsc.getObjectInstance(new
ObjectName("org.openjdk.cookbook:type=SleepProbe"));
            } catch (InstanceNotFoundException e) {
                // bean not found, ignore
            } catch (Exception e) {

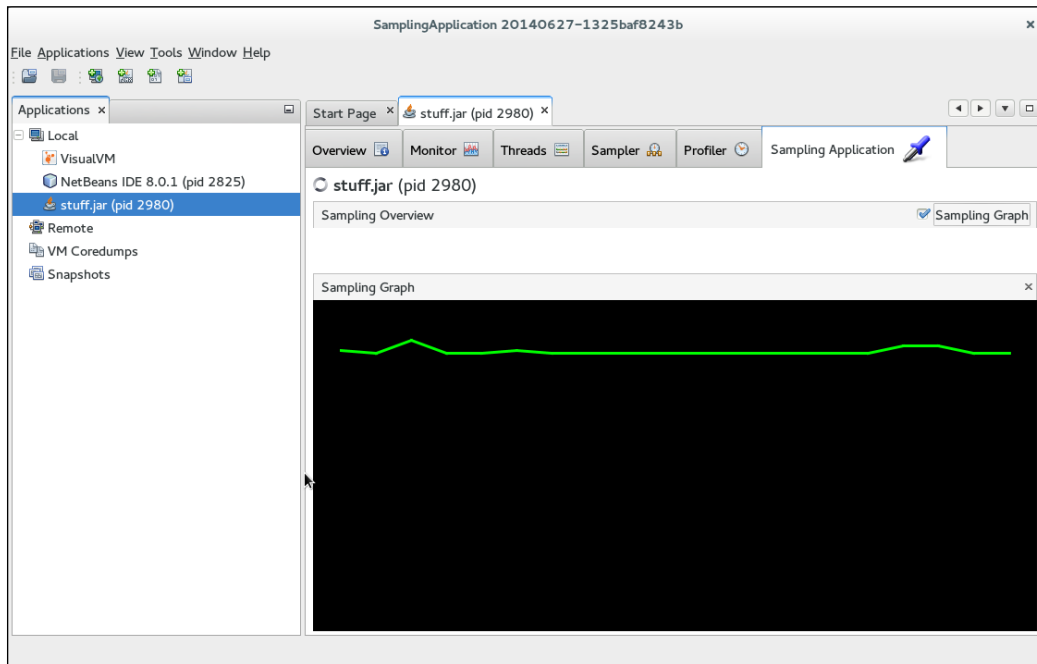
```

```

        Exceptions.printStackTrace(e);
    }
}
}
return instance;
}
}

```

19. The main methods of this class are `supportsViewFor` and `createView`. The `createView` method is small and easy, it just creates a view instance and passes through the application so that the view can get data out of it. The `supportsViewFor` class is slightly bigger, but it doesn't really match. It connects to the given application via JMX and tries to get the instance of `MBean` which our plugin is interested in. If `MBean` is not there, it means that the application is not supported and the method returns `false`.
20. Now it is time to see how the plugin works. To do this, first start the application created on the first step of this receipt. Then right-click on **SamplingModule** and select **Run**. This will start VisualVM with our plugin. From the list of processes in VisualVM, select our process and click on the **Sampling Application** tab. There you will see our graph showing slight changes in the sleep time as shown in the following screenshot:



After following these steps, it should not be a problem for the reader to extend this example and use it for any kind of monitoring applications that he/she wants in VisualVM.

See also

There is some documentation available on the VisualVM website, which helps with creating the plugin for VisualVM and which covers, in more detail, some classes which are used in this receipt, at <http://visualvm.java.net/api-quickstart.html>.

It is also worth having a look at existing plugins source code and some samples available. These can be found in `<code_root>/plugins` and `<code_root>/samples` respectively. For instructions on how to download the source code, please see the *Building VisualVM from the source code* recipe.

Getting benefits from the AdoptOpenJDK project

AdoptOpenJDK is a program, initially developed by a small group of enthusiasts, but which became part of an official OpenJDK community. Some of the purposes are to clarify and simplify OpenJDK building, installation, and usage, but there is still more. It provides build automation, a report generator, build testing, and more. We will cover some basic features that will be useful for everyone.

Getting ready

To follow this recipe, we will need an OpenJDK instance installed, and an Internet connection established.

How to do it...

AdoptOpenJDK is a very helpful and a very sophisticated project set with quite a few different subprojects. Many of them concern testing and evangelism, so we could not include them here as recipes.

Visualizing JIT logs

There is a project named Jitwatch in AdoptOpenJDK. Its purpose is to visualize JIT compiler logs. It helps to find some of our project's performance deficiencies and inspect a native assembly output, as it's just fun after all. Have a look at the following points:

1. First, let's download an executable jar <http://www.chrisnewland.com/images/jitwatch.jar>.
2. In order to analyze an executable, we will need to run it with the following switches:
`-XX:+UnlockDiagnosticVMOptions -XX:+TraceClassLoading`
`-XX:+LogCompilation -XX:+PrintAssembly`

The `UnlockDiagnosticVMOptions` parameter gives access to other diagnostic options, such as `SharedOptimizeColdStart`, `PauseAtStartup`, and so on.

The `LogCompilation` logs compilation activity in detail to `hotspot.log` or `LogFile`, which is yet another VM option.

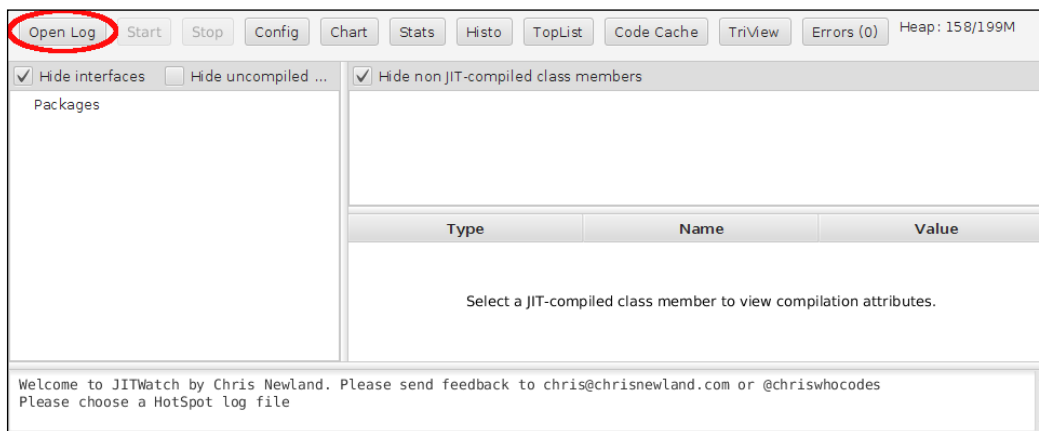
The `TraceClassLoading` parameter lets JVM ensure that all loaded classes are visible, even the ones without any JIT-compiled code related to them.

The `PrintAssembly` parameter lets us see the assembler output of the JIT compilation. It uses `hsdis`, a HotSpot disassembler, which is a part of OpenJDK.

3. Start the analyzer:

```
java -jar ./jitwatch.jar
```

You will see the following screen:



4. Open your `hotspot.log` and, then press **Start**.

Enjoy the experience.

Securing your javadocs

1. We will try to eliminate a javadoc vulnerability (CVE-2013-1571) that seems to exist in Java releases before 7u22.
2. To do that, we will need to clone a repository, such as:

```
git clone https://github.com/AdoptOpenJDK/JavadocUpdaterTool.git
&& cd JavadocUpdaterTool
```

Build the project

```
mvn clean install
```

3. We will try to scan a directory for potential vulnerabilities. We will run the following command from the directory in which the `JavadocPatchTool.jar` file is located:

```
java -jar JavadocPatchTool.jar -R -C <directory>
```

If the tool finds any applicable HTML files, it will print out a list of such files.


4. To fix a single applicable file, we will run the following command:

```
java -jar JavadocPatchTool.jar <path>
```

Here, `<path>` is the path to the directory which contains the applicable file.

5. To fix all the applicable files in the tree, run the following command:

```
java -jar JavadocPatchTool.jar -R <path_to_tree>
```

[ To learn about more benefits from the AdoptOpenJDK project, visit the appropriate page at <https://java.net/projects/adoptopenjdk/pages/Benefits>.]

How it works...

JIT log visualizer is no more than a tool to process logs generated by OpenJDK. But it is a very useful tool that may significantly improve performance.

Java is used in various projects that need security, so the vulnerability in javadocs has an impact on those who still use Java 6 and are unable to change it to Java 7. So the tool that fixes it with no transitions in the technology is really invaluable.

There's more...

There are many more projects in AdoptOpenJDK. Most of them are evangelist or testing ones; thus, they are slightly beyond the scope of this book.

However, you can always find them on various AdoptOpenJDK sites. There is no need to mention all of them in this book, they are tightly linked. In order to find them, it is sufficient to just look around.

9

Testing OpenJDK

In this chapter, we will cover the following topics:

- ▶ Running tests using the downloaded or the built version of jtreg
- ▶ Building jtreg from the source
- ▶ Running the standard set of OpenJDK tests
- ▶ Writing your own test for jtreg
- ▶ Using jtreg in GUI mode
- ▶ Writing TestNG tests for jtreg
- ▶ Compiling JT Harness from the source code
- ▶ Building and running jcstress
- ▶ Writing tests for jcstress
- ▶ Creating a benchmark project using JMH
- ▶ Downloading the source and compiling JMH

Introduction

Back in 1997, when JDK 1.1 was completed and JDK 1.2 had just started, there was an obvious problem—JDK had to be tested somehow and there had to be a tool for it. This is the moment when **jtreg** appeared as the regression test harness. At that time, there were not many testing frameworks for Java. In fact, there was not even much software written in Java. So choices were very limited and the only reasonable option was the framework that was being used at that time for the **Java Compatibility Kit (JCK)**. Its name was **JavaTest**. But, as JCK tests were very different from what a JDK regression test is supposed to do, the framework required some adaptation and that is when jtreg appeared. At the moment, even though many years have passed since 1997, jtreg still remains the primary tool for running unit and regression tests in OpenJDK. Since its introduction in OpenJDK, more than 10,000 tests have been created that run using the jtreg framework.

As a testing tool, jtreg might look a little unusual for modern developers who are used to tools such as JUnit and TestNG. The main reason jtreg looks different is because it has a long history. It appeared before JUnit in 1997, which appeared sometime around 2000. At that time, especially when technology was so young, three years was a long period. It seems possible that the other contributing factor was that jtreg for a long time remained a proprietary tool and, as it was doing its job, there was no need to change it. Also, it was not open to the wide open source community, who could have changed its shape over time. The other reason that contributes to its difference when compared to the current *classical* tools is that it provides some special features that are not usually available in testing frameworks, but are required to do testing on JDK. These features include running a test in a separate instance (or for some tests, even several instances) of JVM with a specific set of parameters, testing Java applets (remember them?), running shell scripts as tests, running GUI tests which require user interaction, and so on. This is a pretty big set of additional features, which is enough to justify building a separate framework for it.

Having said all that, it would not be fair to say that jtreg is some old-fashioned tool that is stuck in the 1990s and does not attempt to change itself to be closer to the modern way of building frameworks for software testing. It has been integrated with testing frameworks such as **TestNG** and provides a way to create tests that are based on that framework. However, the majority of tests in JDK are still just classes with a main method that are executed by the framework. Though, to be fair, such an approach has its benefits since it allows one to run individual tests without any framework. There are also some tests that are just batch files and there is an ongoing effort to get rid of them.

As a tool that evolved from JavaTest, jtreg inherited compatibility with its framework. This compatibility has been now isolated into a separate project called **Java Test Harness (JT Harness)**. This is a framework for running, building, and deploying tests suites. It also provides a GUI to manage and execute the test suites.

In this chapter, you will learn enough to be a confident jtreg user and know how to use JT Harness. You will find out how to build jtreg and JT Harness from the source code, how to run tests, and how to write your own tests. The chapter covers only plain Java and TestNG tests, as they are the most useful ones for OpenJDK developers. This chapter does not cover the usage of shell tests, as their usage is not considered a good practice and they were created to work around JVM limitations that existed a few years back. At the current moment, all OpenJDK contributors are encouraged to replace the shell tests with the Java version, wherever possible.

Running tests using the downloaded or the built version of jtreg

The simplest way to get started with jtreg is to just download it, unpack it, and run some tests. In this recipe, we will do exactly that, without doing any additional things such as building it from the source code or trying to create our own tests.

Getting ready

For this recipe, nothing much is really required—just the Internet connection, a machine with an installed or built OpenJDK, and the OpenJDK source code. In a Windows environment, Cygwin has to be installed in your machine.

How to do it...

The following are a few simple steps to get a set of tests executed by jtreg:

1. If jtreg is not yet available on the machine, go to the official page of jtreg (<https://adopt-openjdk.ci.cloudbees.com/job/jtreg/lastSuccessfulBuild/artifact/>) and download the latest available version of jtreg. The other option is to build it from the source code. To do this, follow the instructions in the *Building Jtreg from the source* recipe that is covered later in this chapter. After you have either downloaded jtreg or built it using the source code, proceed to the next step.
2. Unpack the downloaded archive to a folder.
3. In the root folder of the OpenJDK source tree, create a shell script with the name, `run_test.sh`. This script will be used to run jtreg:

```
#!/bin/sh
export JT_JAVA=/etc/alternatives/java_sdk_1.7.0_openjdk/
/home/user/openjdk/jtreg/bin/jtreg -jdk:$JT_JAVA -agentvm
-automatic -verbose:summary -w build/jtreg/work -r build/jtreg/
report hotspot/test/compiler/5091921
```

The only thing here that needs to be changed is the `JT_JAVA` environment variable that has to point to the version of OpenJDK that is higher or equal to 1.5.

4. After running the script, you will see the output as shown:

```
[user@localhost jdk7u_clean]$ ./run_test.sh
Passed: compiler/5091921/Test5091921.java
Passed: compiler/5091921/Test6186134.java
Passed: compiler/5091921/Test6196102.java
```



```
Passed: compiler/5091921/Test6357214.java
Passed: compiler/5091921/Test6559156.java
Passed: compiler/5091921/Test6753639.java
Passed: compiler/5091921/Test6850611.java
Passed: compiler/5091921/Test6890943.java
Passed: compiler/5091921/Test6897150.java
Passed: compiler/5091921/Test6905845.java
Passed: compiler/5091921/Test6931567.java
Passed: compiler/5091921/Test6935022.java
Passed: compiler/5091921/Test6959129.java
Passed: compiler/5091921/Test6985295.java
Passed: compiler/5091921/Test6992759.java
Passed: compiler/5091921/Test7005594.java
Passed: compiler/5091921/Test7020614.java
Test results: passed: 17
```

5. After the test, run the HTML report that is generated by jtreg. This is available in the folder specified by the `-r` parameter.

How it works...

As you can see, the shell script that is used to run jtreg is simple and there are only a few points that need to be clarified. These are the `JT_JAVA` environment variables and the command line arguments of jtreg.

`JT_JAVA` is one of the two environment variables used by the jtreg shell script. `JT_JAVA` specifies the version of Java that will be used to run the framework, but not the tests. In this recipe, for the sake of simplicity, we used the same version of Java to run jtreg and the tests.

The jtreg command line arguments are all described in detail on the jtreg webpage (<http://openjdk.java.net/jtreg/command-help.html>), so we will cover just a few of them that are used in this recipe:

- ▶ `-jdk`: This argument will generate the JDK that will be used to run tests. Basically, it is the version of Java which is tested by the test run. In our example, we used the version that was installed on the machine. If you want to use the version that was built from the source, the variable should be changed appropriately to point to the output of the build.

- ▶ `-agentvm`: This is the mode when `jtreg` uses the pool of reusable JVMs to run the tests. When a test requires a separate JVM for the run, that JVM is not created, but it is borrowed from the pool of reusable instances. If any parameter is not specified, `jtreg` will recreate a JVM for each test and this will significantly slow down the test run.
- ▶ `-verbose:summary`: This argument specifies the output mode. The `summary` parameter means that it will print just the status and the test name.
- ▶ `-automatic`: This argument means that only the automatic tests that do not require user intervention will be run.
- ▶ `-w`: This argument provides the location of the working directory. This will be used to store class files, and so on.
- ▶ `-r`: This argument provides the reporting directory in which reports are to be stored. To see the report, open the `<reporting directory>/html/report.html` file in any browser.

See also

- ▶ The complete list of command line options is available at <http://openjdk.java.net/jtreg/command-help.html>.
- ▶ Some might find it useful to run tests only for a specific bug and, for that, the `bug:<bug_id>` command-line option can be used. The tests' logfiles can be found in the working directory (specified by the `-w` parameter or in `JTwork` if that parameter is not defined). Logfiles are in text format with a `.jtr` extension. These files contain the test output along with the command line and an exception, if thrown, and are very useful for troubleshooting.

Building jtreg from the source

Apart from downloading `jtreg` as a binary package, there is also an option to download the source code and build `jtreg` from the source. This can be beneficial for developers who might want to make changes in the source code or get the latest fix that has not yet been released as a binary package.

Getting ready

You will need an Internet connection, a machine capable of running make and batch files (Linux or Cygwin), and an installed Mercurial.

How to do it...

The following simple steps will show you how to get the jtreg source code and make the build:

1. Download the sources from <http://hg.openjdk.java.net/code-tools/jtreg>. To do this, just execute the following command that will clone the jtreg source tree in the local folder, jtreg:

```
[user@localhost tmp]$ hg clone http://hg.openjdk.java.net/code-
tools/jtreg
destination directory: jtreg
requesting all changes
adding changesets
adding manifests
adding file changes
added 85 changesets with 1239 changes to 602 files
updating to branch default
586 files updated, 0 files merged, 0 files removed, 0 files
unresolved
```

After the execution of the command, the current directory can be found in the new jtreg folder with all the jtreg sources.

2. If there is no Ant software installed on the machine, install it using yum (or any other packaging tool), or simply download it from <http://ant.apache.org/> and then unpack it. If you are using a Linux machine, you also have an option to install it by running yum or any other similar tool that can be found at <http://ant.apache.org/> and then unpack it.
3. Download the latest available version of JT Harness, which is available at <https://jtharness.java.net/>. Unpack it in the jtreg/lib folder.
4. jtreg requires JUnit, but not the latest version. The version has to be earlier than 4.11. The simplest way to get it would be to download the version from Maven central at <http://mvnrepository.com/artifact/junit/junit/4.5>. A jtreg build requires just the JAR file. Put this file in the jtreg/lib folder.
5. To make things slightly more exciting, a jtreg build also requires another testing framework—**TestNG v.6.8**. This can be downloaded from <http://testng.org/doc/download.html>.

Note that the required version is the version for Ant users. To make things simpler, just use the link, <http://testng.org/testng-6.8.zip>. Unpack the downloaded archive in the jtreg/lib folder.

6. The next dependency is JavaHelp. This seems to be available only via a direct link at http://download.java.net/javadesktop/javahelp/javahelp2_0_05.zip. Unpack it in the `jtreg/lib` folder.
7. And then, finally, the last dependency, that is Xalan, the XML transformation library. The required version is 2.7.1, which can be downloaded from one of the websites listed at <http://www.apache.org/dyn/closer.cgi/xml/xalan-j>. Follow the same procedure that you did with the other libraries and unpack it in the `jtreg/lib` folder.
8. Now, it is time to write the script that will execute the build. Put the following script code into the `make.sh` file of the `jtreg` folder, created in the first step:

```
#!/bin/sh
export JDK15HOME= /etc/alternatives/java_sdk_1.7.0_openjdk/
export JAVAHELP_HOME=/home/user/openjdk/jtreg/lib/jh2.0/javahelp
export ANTHOME=/usr/share/ant
export JTHARNESS_HOME=/home/user/openjdk/jtreg/lib/jharness4.4.1
export JUNIT_JAR=/home/user/openjdk/jtreg/lib/junit4.5/junit-4.5.jar
export TESTNG_HOME=/home/user/openjdk/jtreg/lib/testng-6.8
export TESTNG_JAR=/home/user/openjdk/jtreg/lib/testng-6.8/testng-6.8.jar
export XALANHOME=/home/user/openjdk/jtreg/lib/xalan-j_2_7_1
make -C make
```

As you can see, the script is simple and it is only required to set the environment variables. Not much explanation is required, as all the variable names are pretty self-explanatory. So, just assign appropriate values that are relevant for your machine setup. All the variables are mandatory and have to be defined to run the framework.

9. The final step is to just run that script:

```
[user@localhost jtreg]$ ./make.sh
```

After the build, which should take just a few seconds, the resulting JAR file can be found in the `build/images/jtreg/` folder. This folder will contain a fully workable and self-sufficient `jtreg` distribution:

```
[user@localhost jtreg]$ ls -l build/images/jtreg/
total 60
drwxrwxr-x. 2 user user 4096 May  3 21:27 bin
-rw-rw-r--. 1 user user 994 May  3 21:23 COPYRIGHT
drwxrwxr-x. 3 user user 4096 May  3 21:23 doc
```

```
drwxrwxr-x. 4 user user 4096 May  3 21:27 legal
drwxrwxr-x. 2 user user 4096 May  3 21:27 lib
-rw-rw-r--. 1 user user 19241 May  3 21:27 LICENSE
drwxrwxr-x. 3 user user 4096 May  3 21:23 linux
-rw-rw-r--. 1 user user 3790 May  3 21:27 README
-rw-rw-r--. 1 user user 72 May  3 21:27 release
drwxrwxr-x. 3 user user 4096 May  3 21:23 solaris
drwxrwxr-x. 3 user user 4096 May  3 21:23 win32
```

Running the standard set of OpenJDK tests

This recipe is not much different from the one that describes a simple test execution. However, it will focus on how to run JDK tests. This knowledge might be required if someone is making changes to HotSpot or to any other part of OpenJDK.

Standard tests are available only in three root folders for JDK7 and in four folders for JDK8. These are `hotspot`, `jdk`, `langtools`, and `nashorn` (for `jdk8` only). Although tests are not available for other areas such as CORBA, JDBC, JAXP, and so on, it doesn't mean that they are not tested at all. It just means that tests for them are not part of OpenJDK, that is, they are not provided by vendors.

The way tests are organized varies with the dependency of the area they are testing, for example, `hotspot` and `langtools` are mostly grouped by functional areas they are testing and, then, by bugs (by their numbers). The `jdk` folder is mostly organized by the package name, as this set of tests covers the Java APIs.

Keep in mind that some tests might fail, but it does not mean that something is particularly wrong with OpenJDK. It just means that there are some situations when it is rather hard to create a test that can pass in any environment. For example, there can be tests that need a special network configuration or some other sort of special environment, which might not be set up on the machine.

There is a set of tests that are known to fail and there is usually a good reason for that. The most obvious example is a test that covers some known problem, but whose fix is not going to happen soon. These tests are listed in `jdk/test/ProblemList.txt` or marked with a `@ignore` tag. These tests should usually be excluded from the standard test run.

Getting ready

As we will be running tests for OpenJDK, it is required to have relevant sources available on the machine. The machine should also be set up for the OpenJDK build, as we will be using OpenJDK make files to execute the test run.

jtreg should be downloaded and unpacked in a folder on the machine so that it is ready for the test run.

Some OpenJDK tests are shell scripts, so you will need to use either a Linux machine or, in the case of a Windows machine, Cygwin with support for the Bourne shell, even though Cygwin is not recommended, as there is a chance that some shell tests will not run properly on it.

How to do it...

One of the most convenient ways to run tests is to run them separately for each area, such as `hotspot`, `jdk`, and so on. As the aim of this recipe is to just explain the concept, we will be using the `jdk` tests, which are just a subset of all the tests available in OpenJDK, but the same pattern can be applied to all the other areas as well. Follow the given steps:

1. In the OpenJDK source root folder, create a file with the name, `run_jdk_lang_tests.sh` and the following content:

```
#!/bin/sh
export JTREG_HOME=/home/user/openjdk/jtreg/build/images/jtreg/
export JT_HOME=/home/user/openjdk/jtreg/build/images/jtreg/
export PRODUCT_HOME=/home/stas/openjdk/jdk7u_clean/build/linux-
amd64/j2sdk-image/
cd jdk/test
make TESTDIRS=java/lang
```

The `JTREG_HOME` and `JT_HOME` environment variables are both the same and should point to a folder with `jtreg`. Unfortunately, there are places in `make` files where both these variables are used.

`PRODUCT_HOME` points to a JDK in the test. It is not strictly required to point it to a version of JDK that was just built from the source, but there is also no sense in executing tests on a version that one can't change.

`TESTDIRS` points to a subset of tests to run. Obviously, the wider that subset, the more tests will be executed and the longer it will take to run. So usually it makes sense to limit this subset to something reasonable, unless there is a need to do a regression test after making big changes.

2. Now let's run the script. It will execute tests in the `jdk` folder and output hundreds of lines like these:

```
TEST: java/lang/StringBuilder/Insert.java
      build: 1.112 seconds
      compile: 1.112 seconds
```

```
main: 0.161 seconds
TEST RESULT: Passed. Execution successful
-----
```

3. When it's all finished, the make script will report something as shown:

```
Summary:
TEST STATS: name= run=383 pass=383 fail=0 excluded=4
EXIT CODE: 0
EXIT CODE: 0
Testing completed successfully
```

The preceding output tells us how many tests were executed, how many failed, how many passed, and so on. Now, when all the tests that we were interested in are run, the results can be found in the `jdk/build/linux-amd64/testoutput/JTreport` folder. There will be standard jtrex text and HTML report files that can be viewed using any web browser.

There's more...

If someone came to a point where there was a need to run an OpenJDK test, there is a good chance that there would be a situation when the tests had to be updated or extended. In that case, it is important to have a good understanding of what is going on inside these tests and what is the exact reason for each of them to exist. Mostly, that information is available in the `@bug` and `@summary` tags. It is highly recommended to pay attention to their content and put in some effort to see how they correlate with actual test code.

Most of the tests contain additional information in the `@bug` and `@summary` tags. It is important to refer to these tags to understand the reason for the test. For example, when you run tests, it is not uncommon to see an output like the following:

```
-----
TEST: java/lang/invoke/7157574/Test7157574.java
  build: 1.194 seconds
  compile: 1.193 seconds
  main: 0.199 seconds
TEST RESULT: Passed. Execution successful
```

It means that this is the test for the bug with ID 7157574, which can actually be found in the JDK bug tracking system at <https://bugs.openjdk.java.net/browse/JDK-7157574>. And, when one takes a look at the test, the following information will be in the header:

```
/* @test
 * @bug 7157574
```

```

* @summary method handles returned by reflective lookup API sometimes
have wrong receiver type
*
* @run main Test7157574
*/

```

This header has a reference to the bug that this test is testing and, in the summary section, it explains exactly what this test is doing. Also, when you look at the source of a test, it is quite common to see that it contains a very detailed explanation of the problem and the way that the problem is being tested.

Writing your own test for jtreg

If you are going to add a new feature to OpenJDK or fix a bug, it is really a good idea to have a test case to cover the change in functionality and to ensure that the implementation change doesn't break anything. This recipe will help you to get some understanding of the process and create a simple test. You will find that writing your own test case for jtreg is not a complicated task, but it can be slightly unusual in some respects.

Getting ready

All that is required for this recipe is an installed jtreg and the OpenJDK sources. The latter is required only because this recipe assumes that the newly created test is for OpenJDK.

How to do it...

Writing tests for jtreg can be a little unusual, but when you get used to the pattern it is actually quite easy. To begin, just follow the steps. Keep in mind that all the paths are given relative to the OpenJDK source root:

1. Go to the root folder of OpenJDK and create the `jdk/test/demo/SampleTest.java` file first:

```

/* @test
 * @summary Test to ensure that computer wasn't moved to the past
 * @compile SampleTimeProvider.java
 * @run main SampleTest
 * @run main/othervm SampleTest
 */
public class SampleTest {
    public static void main(String[] args) {
        long currentTime = new SampleTimeProvider().
getCurrentTime();

```



```
        if ( currentTime < 0 ) {  
            throw new RuntimeException("It can't be 1969!");  
        }  
    }  
}
```

2. Then, create the `jdk/test/demo/SampleTimeProvider.java` file as follows:

```
public class SampleTimeProvider {  
    public long getCurrentTime() {  
        return System.currentTimeMillis();  
    }  
}
```

3. Now, create a script file with the name, `run_jtreg.sh` in the source root folder of JDK and run it:

```
#!/bin/sh  
  
export JT_JAVA=/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.60-  
2.4.7.0.fc20.x86_64  
  
/home/user/openjdk/jtreg/build/images/jtreg/bin/jtreg -jdk:$JT_  
JAVA -agentvm -automatic -verbose:summary -w build/jtreg/work -r  
build/jtreg/report -exclude:./jdk/test/ProblemList.txt jdk/test/  
demo/SampleTest.java
```

4. The output should be as follows:

```
[user@localhost jdk7u]$ ./run_jtreg.sh  
Directory "build/jtreg/work" not found: creating  
Directory "build/jtreg/report" not found: creating  
Passed: demo/SampleTest.java  
Test results: passed: 1  
Report written to /home/user/openjdk/jdk7u/build/jtreg/report/  
html/report.html  
Results written to /home/user/openjdk/jdk7u/build/jtreg/work
```

As you can see, only one test was run and the run was successful. So, writing a simple test case is a very simple task.

5. Now, the final step. After all the tests have been run, let's look at the test result whose path was provided in the output of `jtreg`. Let's open `report/html/repost.html` in a web browser and take a look at what's there:

Configuration and Other Settings

Test suite [/home/stas/openjdk/jdk7u/jdk/test](#)

Standard Values

Tests	demo/SampleTest.java
Prior Status	(Any Status - Pass, Fail, Error, Not Run)
Excluded Tests	108 exclude total exclusions /home/stas/openjdk/jdk7u/jdk/test/ProblemList.txt
Concurrency	1
Timeout factor	1
Keywords	!manual

How to run the tests

Configuration Log [regtest](#)

Environment [regtest](#)

Where to put the results:

Work directory [/home/stas/openjdk/jdk7u/build/jtreg/work](#)

Report directory [/home/stas/openjdk/jdk7u/build/jtreg/report/html](#)

Results

Tests that passed	1	plain view	grouped view
Tests that were not run	4575	plain view	
Total	4576		

Here, we can see that only one test, **demo/SampleTest.java**, was executed and it was the one that passed.

How it works...

Some explanation is required to find out what actually happened. The test itself is in the `SampleTest.java` file. `jtreg` knows that the file contains the test by the presence of the `@test` tag in the class header comments. Without that tag, `jtreg` will not consider it as a test.

The only purpose of the `@summary` tag is to give a summary description for the test. This description will also be used in logs and reports. It is very important to have a good, readable description for that tag. Also, if a test is for a bug, it is necessary to have the `@bug` tag populated with an appropriate bug number.

The next tag, `@compile`, has a reference to another file, which is required to be compiled to run the test. The only reason for the existence of `SampleTimeProvider.java` is to show how to use the `@compile` tag. It is very unusual for Java to do things like this. Usually, everything is compiled and then things are picked from the classpath, but this is the way Java works.

The `@run` tag tells the harness how to run the test. As can be seen from the test class, this parameter can be defined several times, which means that the test will also be executed several times and each time it will be run with a configuration defined by the associated run tag. In our example, there are two runs, one in the same VM and the other in a new instance of VM, which is specified by the `othervm` parameter. If this tag is not defined, then, by default, `jtreg` assumes that it is `@run main ClassName`.

Note that a test fails if it runs for longer than 2 minutes (120 seconds) and can be overwritten by `@run main/timeout=xxx`.

Usually, a test indicates its failure by throwing an exception. A test from this recipe will throw `RuntimeException` when its conditions are not met.

`jtreg` requires the `TEST.ROOT` file to be created in the tests' root folder. Without that file, it will not execute any test. Luckily for us, JDK already has the required file with the appropriate content, so there is no need for us to worry about it.

See also

`jtreg` test cases are defined by the Javadoc tags and it is useful to be familiar with all of them. The complete list of tags and information about each tag is available either in the help file that can be accessed by running the `jtreg` command, `-onlineHelp`, or online at <http://openjdk.java.net/jtreg/tag-spec.txt>.

Using jtreg in GUI mode

`jtreg` is not only a command-line tool, but it also provides a relatively sophisticated graphical interface that allows you to run an individual set of tests, prepare test runs, see results of the run, and so on. This recipe will cover some basic features of UI that are enough for the user to start using the tool.

Getting ready

All that is required for this recipe is an installed `jtreg` and the OpenJDK sources.

How to do it...

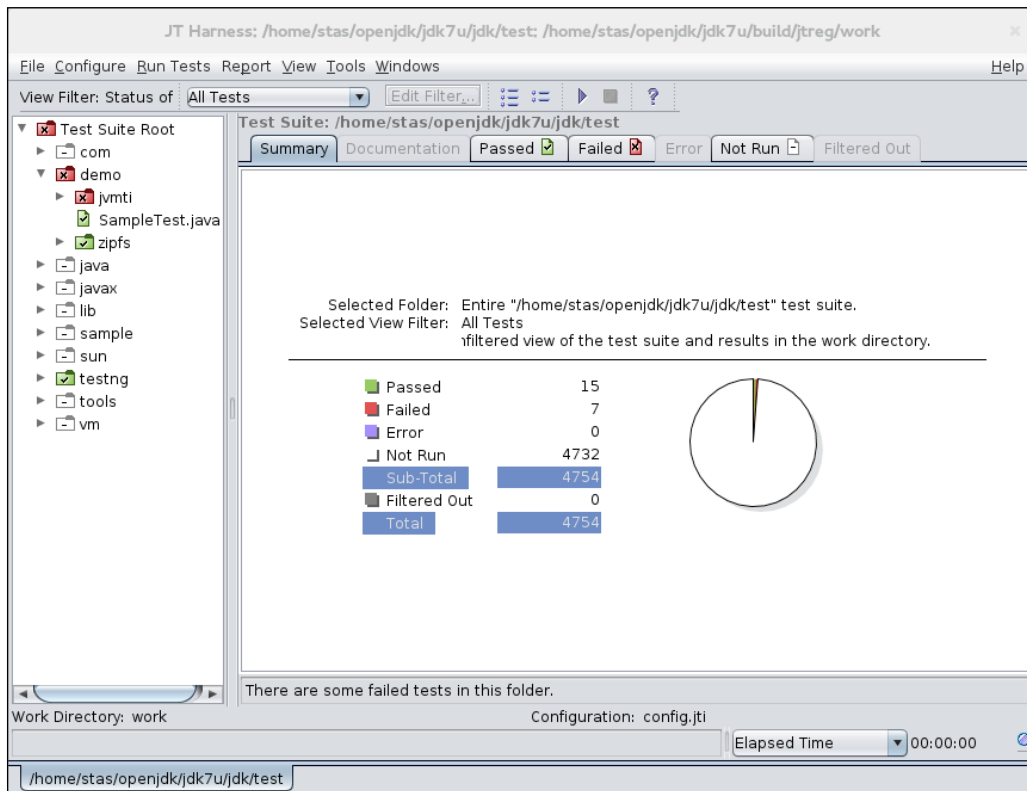
1. Create the following script that will launch jtreg with a graphical UI:

```
#!/bin/sh

export JT_JAVA=/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.60-2.4.7.0.fc20.x86_64

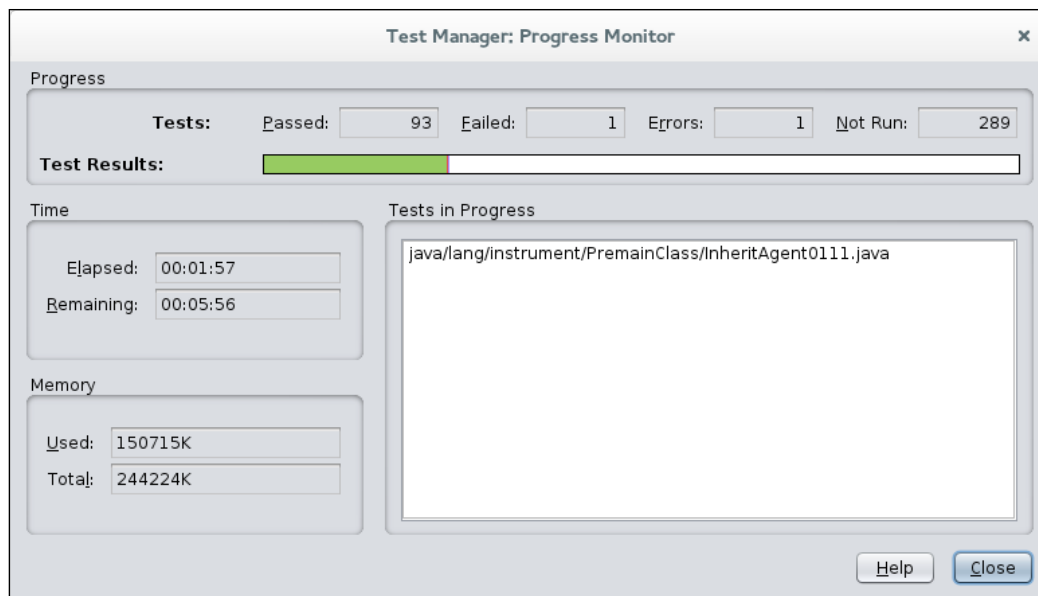
/home/user/openjdk/jtreg/build/images/jtreg/bin/jtreg -g -jdk:$JT_JAVA -agentvm -automatic -verbose:summary -w build/jtreg/work -r build/jtreg/report -exclude:./jdk/test/ProblemList.txt jdk/test/
```

The parameter that tells jtreg to launch JT Harness UI is `-g`. After it is launched, jtreg displays a window similar to the following one:



2. On the right-hand side of the preceding screenshot, you can see the results of the latest test run. It knows where to pick them up from the `-r` parameter. In this screenshot, you can also see the number of successful and failed tests, the total number of executed tests, and some other stats.

- On the left-hand side of the preceding screenshot, there is a tree with all the available tests. This shows all the tests from the root of the test bundle, which is the folder with the `TEST.ROOT` configuration file. The green folder icon indicates the tests that ran successfully and the red one indicates the ones that failed.
- To run a specific test from a set of tests, right-click on the individual test folder and select the **Execute these test** item from the pop-up menu. This will trigger the test run and a new report will be generated. For long running tests, there is a status window, which is available via the **Run Tests | Monitor Progress** menu item:



How it works...

The UI that is used by jtreg does not actually belong to jtreg. It is provided by JT Harness and jtreg just integrates it using the provided plugin system.

JT Harness provides a rich interface not only to run tests, but it also has a set of wizards to create test configurations, various report conversion tools, an agent monitoring tool, and so on. To get more information about all these features, refer to JT Harness online help, which is available via the **Help/Online Help** menu item.

Writing TestNG tests for jtreg

jtreg also provides support to run TestNG test cases, which might be a more familiar technique for many developers. There are some steps that need to be performed to make it happen and this recipe will go through them.

Getting ready

This recipe will require the OpenJDK source code and an installed version of jtreg. On a Windows machine, it is also required to have Cygwin installed.

How to do it...

The following procedure should be more familiar to Java developers than writing a native jtreg test, as TestNG is what most developers have heard of and used. Now, let's get to the practical part and create a test. The paths are relative to the OpenJDK source root.

1. Relative to the OpenJDK root folder, create a folder with the name, `jdk/test/testng/org/demo`. This is the folder where we are going to create our tests.
2. Create the file, `jdk/test/testng/TEST.properties`, relative to the OpenJDK source root. Add the following line there:

```
TestNG.dirs=.
```

It is not strictly required to create the file, and it is also possible to define the property in `jdk/test/TEST.ROOT`, which will work in the same way as our previous property. However, in most cases, it is practical to have that file so that it can contain some additional configuration that is specific to the set of TestNG tests, for example, the `lib.dirs` property.

3. In the `jdk/test/testng/org/mydemo` folder, create the following file with the name, `MyTestNGTest.java`:

```
package org.mydemo;

import org.testng.Assert;
import org.testng.annotations.Test;

public class MyTestNGTest {
    @Test
    public void testMoreNanosThanMillis() {
        final long nanos = System.nanoTime();
        final long millis = System.currentTimeMillis();
        Assert.assertTrue(nanos > millis);
    }
}
```

4. Now, in OpenJDK's source root folder, create the following bash script (fix the paths as required to match the machine environment):

```
#!/bin/sh

export JT_JAVA=/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.60-2.4.7.0.fc20.x86_64

/home/user/openjdk/jtreg/build/images/jtreg/bin/jtreg -jdk:$JT_JAVA -agentvm -automatic -verbose:summary -w build/jtreg/work -r build/jtreg/report -exclude:./jdk/test/ProblemList.txt jdk/test/testng
```

5. Then, give it a name, `./run_jtreg.sh`, and make it executable (running `chmod +x ./run_jtreg.sh` will do the job) and run it. The resulting output of the script should look similar to the following:

```
[user@localhost jdk7u]$ ./run_jtreg.sh
Passed: testng/org/mydemo/MyTestNGTest.java
Test results: passed: 1
Report written to /home/user/openjdk/jdk7u/build/jtreg/report/html/report.html
Results written to /home/user/openjdk/jdk7u/build/jtreg/work
```

And that's it. The test has been run and, as can be seen from the output, it has passed. The result of the run can be seen by opening `/home/user/openjdk/jdk7u/build/jtreg/report/html/report.html` in the browser application.

How it works...

The file named `TEST.properties` that is created in the folder with the tests root package, requires a bit of explanation. This is the file that contains sets of configurations specific to the TestNG tests in the folder where the file is located. For example, it can have a reference to folders with libraries via the `lib.dirs` property. That is the `path` type property, which is the same as `TestNG.dirs`, and such properties are space-separated lists of paths to some of the folders or files. If the path in the list starts with `/`, then it is evaluated relative to a folder with `TEST.ROOT`, otherwise it is evaluated from the directory containing `TEST.properties`.

See also

As it is possible to use the TestNG annotations within the test classes, it is worth checking the TestNG website, which contains some documentation on this topic at <http://testng.org/doc/documentation-main.html>.

It is also possible to use the jtreg style tags and the TestNG style test together. In that case, the test should be created as a normal jtreg test (see the *Writing your own test for jtreg* recipe), using `testng` as the argument for `@run`, for example, `@run testng SampleTest`. In this scenario, there is no need for `TEST.properties` and things like `lib.dirs` are defined in the test source file via the jtreg tags, rather than in a separate configuration file.

Compiling JT Harness from the source code

JT Harness is a framework that allows you to execute different sets of tests. It is not necessary to use it only with jtreg. Other testing frameworks can be integrated with it as well. This means that it could be useful to have its source code to be able to build it from the source. This is exactly what this recipe is going to explain.

Getting ready

You will need Internet access to download the source code and install the additional software that is required for the build.

Ensure that Ant version 1.6.1 or later is installed on the machine. If not, then install it. The *How to do it...* section depends on the OS you are using. For example, on Fedora, that will be:

```
yum install ant
```

On Windows, the simplest way would be to just download the distribution and add Ant's `bin` folder to the `PATH` environment variable. The Ant distribution can be found at <http://ant.apache.org/>. Keep in mind that, to make it work, the `JAVA_HOME` environment variable has to contain the correct path of the Java distribution.

Ensure that Subversion is installed on the machine. In contrast to other tools, JT Harness doesn't use the Mercurial repository for its source code. On Fedora, Subversion can be installed by running `yum`:

```
yum install svn
```

On any other OS, it really depends on the OS. Check <http://subversion.apache.org/> to find out what is available.

How to do it...

Run the following command to check the source code:

```
svn checkout https://svn.java.net/svn/jtharness~svn/trunk jtharness
```

This command will create a folder with the name, `jtharness`, and download the source code. In that folder, create another folder and name it, `lib`. Here we will put the libraries required to build JT Harness.

Now download the following software (all paths are given relative to the `jtharness` folder):

1. JavaHelp seems to be available only via a direct link from http://download.java.net/javadesktop/javahelp/javahelp2_0_05.zip. Unpack it into `lib/jh2.0`.
2. Go to <http://asm.ow2.org/> and download the ASM Java bytecode manipulation library binaries version 3.1. Unpack the archive into `lib/asm-3.1`. This is required only for compilation.
3. Next, the required library is Java Communications API. Go to <http://www.oracle.com/technetwork/java/javasebusiness/downloads/java-archive-downloads-misc-419423.html> and download Java Communications API 3.0u1 or any other higher version that is available. Unpack the archive into `lib/commapi`. This is required only for compilation.
4. Java Servlet APIs are only required for compilation. Probably the simplest way to download them is to get the JAR from Maven Central. Go to <http://search.maven.org/> and search for `javax.servlet servlet-api`. Download the JAR v.3.1.0 and put it directly into the `lib` folder.
5. The last one is JUnit, which is also only required for compilation. The recommended version is 4.4. Go to <http://junit.org/> and click on the **Download** link to download the appropriate version of the JAR file. Place it directly into the `lib` folder.
6. The next step is to get the sources. To do this, run the following command that will get the latest sources from the trunk:

```
svn checkout https://svn.java.net/svn/jtharness~svn/trunk
jtharness
```

This will create the folder, `jtharness`, with two subfolders: `www` and `code`. The `www` folder contains pages with links to the documentation and other useful information, but we are really interested in the folder with the name `code`. It is actually not strictly necessary to download `www` at all, but it doesn't do any harm either.

7. Go to the folder, `jtharness/code/build`, locate the file `local.properties`, and edit it to set the following properties to point to the appropriate locations:
 - `jhalljar`: This provides the path to the JavaHelp `jhalljar.jar` file
 - `jhjar`: This property provides the path to the JavaHelp `jhjar.jar` file
 - `jcommjar`: This provides the path to `comm.jar` from the Java Communications API
 - `servletjar`: This provides the path to the `servlet-api.jar` file from Java Servlet API
 - `bytecodelib`: This provides a colon-separated path to the `asm-3.1.jar` and `asm-commons-3.1.jar` files
 - `junitlib`: This provides the path to `junit-4.4.jar` from JUnit

After the editing, the file should look similar to this:

```
#Please specify location of jhall.jar here - for compilation
jhalljar = /home/user/openjdk/jtharness/lib/jh2.0/javahelp/lib/
jhall.jar
# needed only at runtime
jhjar = /home/user/openjdk/jtharness/lib/jh2.0/javahelp/lib/jh.jar
# location of jar with implementation of java serial
communications API
jcommjar = /home/user/openjdk/jtharness/lib/commapi/jar/comm.jar
# location of jar with servlet API implementation
servletjar = /home/user/openjdk/jtharness/lib/javax.servlet-api-
3.1.0.jar
# bytecode library (BCEL or ASM)
# these are not interchangeable
bytecodelib = /home/user/openjdk/jtharness/lib/asm-3.1/lib/asm-
3.1.jar:/home/stas/openjdk/jtharness/lib/asm-3.1/lib/asm-commons-
3.1.jar
# JUnit Library - Version 4 currently used to compile 3 and 4
support
junitlib = /home/user/openjdk/jtharness/lib/junit-4.4.jar
# Please specify location where the build distribution (output)
will be created
BUILD_DIR = ../JTHarness-build
```

If it is required, the BUILD_DIR variable can be changed to a different folder but, usually it is not necessary.

8. Now ensure that the current folder is jtharness/code/build and run Ant:

```
[stas@localhost build]$ ant
Buildfile: /home/user/openjdk/jtharness/code/build/build.xml
...skipped...
BUILD SUCCESSFUL
Total time: 45 seconds
When build is finished
```

When the build is finished, the folder, jtharness/JTHarness-build/binaries, contains the distribution of JT Harness.

See also

There are other targets available in the JT Harness build file that you may find useful:

- ▶ `run`: This builds and runs JT Harness. This is not the only way to launch the application. The other option is to run the following command from the source root folder after the build:

```
java -jar JTHarness-build/binaries/lib/javatest.jar
```
- ▶ `clean`: This just builds the distribution directory.
- ▶ `build`: This builds JT Harness and then runs all the tests.
- ▶ `test`: This just runs the test.
- ▶ `Javadoc`: This generates the Javadoc API documentation.
- ▶ `build-examples`: This builds the example test suites packaged with the source. This target also automatically builds the core harness first.

Building and running jcstress

The **Java Concurrency Stress Test (jcstress)** is a set of tests for testing the correctness of the Java concurrency support. This is a new tool that is mostly targeted to Java 8, which means that not all the tests will run on previous versions of Java. As a new tool, jcstress is in its alpha phase and changes for fixes and improvements are common, which means that whoever is working with it is expected to update the source code and rebuild the tool relatively often.

Testing concurrency is not an easy task and it may or may not be easy to make such tests fail even with incorrect code. This happens due to the nature of concurrent code, which might work differently on different hardware configurations. Such variations arise from the number of CPUs or the CPU architecture. Overall, this means that many tests in jcstress are uncertain and they might require a long time before exposing potential problems.

Getting ready

You will need Internet access and the Mercurial repository to download the source code. As jcstress requires Java 8 for compilation and to run the full set of tests, it has to be installed on the machine and set as the current version. This means that the following commands should, as illustrated next, show Java 1.8 as the major version:

```
[user@localhost]$ java -version
openjdk version "1.8.0_11"
OpenJDK Runtime Environment (build 1.8.0_11-b12)
```

OpenJDK 64-Bit Server VM (build 25.11-b02, mixed mode)

```
[user@localhost jcstress]$ javac -version
javac 1.8.0_11
```

The build will also require Maven (one of the Java build tools) to be installed on the machine. The installation of this tool depends on the OS. For example, on Fedora, this can be done by running the following command as root:

```
[root@localhost ~]# yum install maven
```

On other OSs, the simplest way might be to download the binary from <http://maven.apache.org/download.cgi>, unpack the archive, point `M2_HOME` to the root of the unpacked folder and add `M2_HOME/bin` to the path.

How to do it...

Now it's time for action. The first few steps of this recipe will cover the build process and then it will switch to run the actual tests:

1. The first step is to download the sources. Go to the folder where you want to store the source code and run the following command:

```
[user@localhost ~]$ hg clone http://hg.openjdk.java.net/code-tools/jcstress/ jcstress
```

When this command is run, Mercurial is downloaded along with the source code from the remote repository and it is stored in the folder, `jcstress`.

2. Now, to build the tool, move to the folder `jcstress` and run the following command:

```
[user@localhost jcstress]$ mvn clean install -pl tests-custom -am
```

If successful, it should display something like this as the end:

```
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 1:31.300s
[INFO] Finished at: Tue Jul 29 22:23:19 BST 2014
[INFO] Final Memory: 37M/410M
[INFO] -----
```

This means that the build has done what it was supposed to do and the tests are ready to be run.

3. To run all the tests, use the following command:

```
java -jar tests-custom/target/jcstress.jar
```

This will output thousands of lines like these:

```
(ETA: 00:39:58) (R: 1.36E+07) (T: 46/898) (F: 1/1) (I: 3/5)
[OK] o.o.j.t.atomicity.primitives.plain.VolatileCharAtomicityTest
(ETA: 00:40:05) (R: 1.36E+07) (T: 46/898) (F: 1/1)
(I: 3/5) [OK] o.o.j.t.atomicity.primitives.plain.
VolatileDoubleAtomicityTest
```

ETA is the estimated completion time, R is the runtime in nanoseconds, T is the test number, F is the fork number, and I is the test iteration number. This is all followed by the result (OK in this example) and the full name of the test class.

As you can see, the full standard set of tests run for approximately 40 minutes, and this might be too much, so there is an option to select which test to run using the `-t` parameter. This is a regular expression selected for the tests, for example:

```
[user@localhost jcstress]$ java -jar tests-custom/target/jcstress.jar -t
".*ByteBufferAtomicityTests.*"
```

This will only run tests which have `ByteBufferAtomicityTests` anywhere in their name.

When the tests are completed, it is time to have a look at the reports that are generated and, by default, they are put into the `./results/` folder. There you can find the file, `index.html`, which can be opened with any browser. The result will list all the tests and, if you click on the tests, all the observer outputs. These outputs can be something that is expected, not expected, or expected but somehow surprising to the user. This can result in the following outcomes for a test:

- ▶ **FAILED:** The test failed and the outcomes were not what was expected
- ▶ **ERROR:** The test crashed
- ▶ **ACCEPTABLE:** The test results matched the specifications
- ▶ **ACCEPTABLE_INTERESTING:** This is the same as **ACCEPTABLE**, but it has something to highlight
- ▶ **ACCEPTABLE_SPEC:** This is the same as **ACCEPTABLE**, but some interesting behavior is also observed that might not be otherwise expected

There's more...

It is recommended that you have a look at the other command-line options that are available for `jcstress`. This information can be retrieved by running the following command:

```
[user@localhost jcstress]$ java -jar tests-custom/target/jcstress.jar -h
```

jcstress has its own page at <http://openjdk.java.net/>, which has some very useful information and links to the source and mailing list at <http://openjdk.java.net/projects/code-tools/jcstress/>.

Writing tests for jcstress

The **Java Concurrency Stress test** is a wonderful tool that is used by JDK authors to ensure that their concurrent code works correctly with regards to concurrency. Concurrent code is hard to write and even harder to test. Most of their tests are probabilistic, require a lot of skill to write, and might take many days running time and the appropriate hardware to exhibit the fail behavior. Considering all this complexity, having a framework that can help with correct test execution is a big advantage. This recipe will go through the steps that are required to write your own tests for jcstress.

Getting ready

To follow this recipe, the only requirement is to have an environment that is capable of compiling and running jcstress (see the *Building and running jcstress* recipe).

How to do it...

The following steps will lead you through the process of creating a test and running it using jcstress:

1. To start, we will need a code to test. Let's pick the problem which, arguably, is the most common cause of headaches in concurrency and is very easy to reproduce. Data race sounds like a good candidate. We will make a class called `CASValue` and implement it:

```
package org.openjdk.jcstress.tests;

public class CASValue {
    private int i = 0;
    public boolean cas(int expected, int newValue) {
        boolean res = false;
        if (i == expected) {
            i = newValue;
            res = true;
        }
        return res;
    }
}
```

This class implements a single operation which is supposed to perform the compare-and-swap operation (see <http://en.wikipedia.org/wiki/Compare-and-swap>). Without synchronization, it will not work correctly in a multithreaded environment and should fail the test that we will create in the next steps of this recipe.

2. In the source root folder of jcstress, create the file, `tests-custom/src/main/java/org/openjdk/jcstress/tests/CASValue.java`, and put the source code of the `CASValue` class in it.
3. Now it's time to write a test to see whether our implementation is wrong. The test class will look like this:

```
package org.openjdk.jcstress.tests;

import org.openjdk.jcstress.annotations.*;
import org.openjdk.jcstress.infra.results.LongResult2;

public class CASValueTests {
    @State
    public static class S extends CASValue { }

    @JCStressTest
    @Description("Tests correctness of CASValue CAS operations.")
    @Outcome(id = "[5,2]", expect = Expect.ACCEPTABLE, desc = "T1
-> T2 execution.")
    @Outcome(id = "[1,10]", expect = Expect.ACCEPTABLE, desc = "T1
-> T2 execution.")
    public static class ValCas_ValCas {
        @Actor public void actor1(S s, LongResult2 r) {
            r.r1 = s.cas(0, 5) ? 5 : 1;
        }
        @Actor public void actor2(S s, LongResult2 r) {
            r.r2 = s.cas(0, 10) ? 10 : 2;
        }
    }
}
```

4. Save this file in the same folder as `CASValue.java`, that is, in `tests-custom/src/main/java/org/openjdk/jcstress/tests/`, and give it the name, `CASValueTests.java`.
5. The `CASValueTests` class is the container class for other classes within the test. This is not strictly required, but it helps to keep the code clean. The `ValCas_ValCas` class, which is annotated with `@JCStressTest`, is the test case class that contains two actors—methods annotated with `@Actor`. These are the methods that will be run in parallel by the test framework.

6. The subclass `S`, annotated with `@State`, is the state that was shared among the actors and, in this case, is the class under test. It extends our class, `CASValue`, and is created solely to avoid adding the `@State` annotation on `CASValue`.
7. The `@Outcome` annotations specify the outcomes of the test. The outcomes can be `ACCEPTABLE`, `FORBIDDEN`, `ACCEPTABLE_INTERESTING`, and `ACCEPTABLE_SPEC`. These are defined by the `expect` attribute. The `id` attribute provides the list of outcomes, whereas `desc` is just a description of the outcome. This test case specifies that, for our test, the only valid outcomes for the values in `LongResul2` are 5 and 2 and 1 and 10, which are the only expected ones if CAS works as expected. Any other outcomes are forbidden and will cause the test case to fail, which is exactly what we want.
8. Now it's time to compile the test. To do this, run the following command from the source root directory of `jcstress`:

```
[user@localhost jcstress] mvn clean install -pl tests-custom -am
```

This will just run the build for the `tests-custom` project, which will compile the classes that we have just created.

9. The next step is to run our test and see if it works:

```
[user@localhost jcstress] java -jar tests-custom/target/jcstress.jar -t ".*CASValueTests.*"
```

The `-t` parameter in this command specifies that we only want to run the tests that satisfy the `.*CASValueTests.*` regular expression.

As expected, the tests should fail, as the provided implementation does implement the CAS operation correctly. The output should have several test outcomes similar to this one:

```
(ETA:          n/a) (R: 5.95E+08) (T:   1/1) (F: 1/1) (I: 1/5)
[FAILED] o.o.j.t.CASValueTests$ValCas_ValCas
Observed state  Occurrences Expectation Interpretation
[5, 2]   (3,230,666)          ACCEPTABLE T1 -> T2 execution.
[1, 10]  (2,613,825)          ACCEPTABLE T2 -> T1 execution.
[5, 10]  (7,609,449)          FORBIDDEN Other cases are not expected.
```

It shows how many outputs of each outcome were observed. So far, the incorrect outcome `[5, 10]` is the leader with 7,609,449 occurrences. This shows that the test worked correctly and helped us to identify that we need to fix our implementation on the CAS class.

10. Now let's fix our class and run the test again. The easiest, though not the most efficient, way to fix the problem is to simply add the `synchronized` modifier to our CAS method:

```
public class CASValue {  
    private int i = 0;  
    public synchronized boolean cas(int expected, int newValue) {  
        boolean res = false;  
        if (i == expected) {  
            i = newValue;  
            res = true;  
        }  
        return res;  
    }  
}
```

11. After changing the implementation, run the build again:

```
mvn clean install -pl tests-custom -am
```

12. Then, rerun the test:

```
java -jar tests-custom/target/jcstress.jar -t ".*CASValueTests.*"
```

13. Now, the test should not show any failures and report that the test runs were successful:

```
(ETA: n/a) (R: 3.35E+08) (T: 1/1) (F: 1/1) (I: 1/5) [OK]  
o.o.j.t.CASValueTests$ValCas_ValCas  
(ETA: 00:00:02) (R: 1.69E+07) (T: 1/1) (F: 1/1) (I: 2/5) [OK]  
o.o.j.t.CASValueTests$ValCas_ValCas  
(ETA: 00:00:01) (R: 1.22E+07) (T: 1/1) (F: 1/1) (I: 3/5) [OK]  
o.o.j.t.CASValueTests$ValCas_ValCas  
(ETA: 00:00:00) (R: 1.07E+07) (T: 1/1) (F: 1/1) (I: 4/5) [OK]  
o.o.j.t.CASValueTests$ValCas_ValCas  
(ETA: now) (R: 1.00E+07) (T: 1/1) (F: 1/1) (I: 5/5) [OK]  
o.o.j.t.CASValueTests$ValCas_ValCas
```

This shows that the change in the implementation worked and the implementation is correct according to the test case.

By following these steps, a developer can ensure that a concurrent code works as expected according to the specification, as long as the tests and test cases are implemented and defined correctly. However, keep in mind that concurrency is something that is hard to test, and an implementation that works on one hardware can easily fail on other hardware. It means that it is recommended to run these test on the widest possible range of configurations.

See also

This framework is built and maintained by Aleksey Shipilëv who has his own blog and also delivers lectures in various conferences. We recommend that you visit his home page (<http://shipilev.net/>), watch his videos on YouTube (for example, <https://www.youtube.com/watch?v=4p4vL6EhzOk>), and read some of his papers. This will help you to get tons of information on correct concurrency testing, concurrency in general, support for concurrency in Java, and other related topics.

Creating a benchmark project using JMH

Micro benchmarking itself is not an easy topic, and doing it correctly using languages like Java is a very difficult task. These difficulties arise from the way Java executes the code and the infrastructure required by JVM. Just as things like JIT and GC may affect the results of micro benchmarking heavily, to ensure that the result of each run is consistent and correct might not be an easy task to accomplish. To help with this problem, there are several frameworks that can help to ensure that the benchmark test runs properly. One of these frameworks is **Java Microbenchmark Harness (JMH)**, which is a part of OpenJDK. This recipe will explain how developers can use this framework to benchmark his/her own code.

Getting ready

This recipe requires a machine with an Internet connection, Maven, Java SDK, and your favorite IDE that has support for Maven projects.

How to do it...

The following steps will take you through the process of creating a benchmark project and writing the benchmark, which can be used to analyze the performance of the code:

1. In the command line, run the following Maven command:

```
[user@localhost ~] mvn archetype:generate -DinteractiveMode=false
-DarchetypeGroupId=org.openjdk.jmh -DarchetypeArtifactId=jmh-
java-benchmark-archetype -DgroupId=org.benchmark
-DartifactId=mybenchmark -Dversion=1.0
```

2. After running this command, in the current directory, Maven will create a folder with the name, `mybenchmark`, which will have the skeleton of the project. If all goes well, the build should end with an output that is similar to the following:

```
[INFO] project created from Archetype in dir: /home/user/openjdk/
mybenchmark
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 9.188s
[INFO] Finished at: Sat Aug 02 20:16:01 BST 2014
[INFO] Final Memory: 11M/129M
[INFO] -----
```

3. Now, when the project is generated, we can start working with it and create our first microbenchmark test. Open the generated project file (`/home/user/openjdk/mybenchmark/pom.xml`) with your favorite IDE. Ensure that Maven is correctly configured and all dependencies are downloaded correctly. Notice that there is already a class created for the benchmark whose name is `org.benchmark.MyBenchmark`. At the beginning, all it has is a single method where we will later put the code we are going to test.
4. As an example, let's test something that is relatively simple but has some room for improvement. A binary search is a good choice for this purpose. So, let's draft a simple implementation and put it into the `org.benchmark.BinarySearch1` class as shown next:

```
package org.benchmark;
public class BinarySearch1 {
    public static int search(long[] arr, long value) {
        return search(arr, value, 0, arr.length-1);
    }

    private static int search(long[] arr, long value, int start,
int end) {
        if (end < start)
            return -1;

        int imid = start + ((end - start) / 2);
        if (arr[imid] > value)
            return search(arr, value, start, imid-1);
        else if (arr[imid] < value)
            return search(arr, value, imid+1, end);
```

```

        else
            return imid;
    }

```

This is a very basic implementation, which will be fine for our experiment. If you are not familiar with binary search or where to get more information about this algorithm, visit the Wikipedia page at http://en.wikipedia.org/wiki/Binary_search_algorithm.

- Now, when the first draft of the implementation is ready, we will create a microbenchmark for it. Put the following code into the `org.benchmark.MyBenchmark` class:

```

package org.benchmark;

import org.openjdk.jmh.annotations.*;
import java.util.Arrays;
import java.util.concurrent.TimeUnit;

@State(value = Scope.Thread)
@BenchmarkMode (Mode.AverageTime)
@OutputTimeUnit (TimeUnit.NANOSECONDS)
public class MyBenchmark {
    private long[] data = new long[5000];

    @Setup
    public void setup() {
        for (int i = 0; i != data.length; ++i) {
            data[i]=(long) (Math.random()*Long.MAX_VALUE-1);
        }
        Arrays.sort(data);
    }

    @Benchmark
    public int testBinarySearch1 () {
        return BinarySearch1.search(data, Long.MAX_VALUE);
    }
}

```

This code requires some clarification. The `@State` annotation is required to tell JMH that this class contains some data that is used by the tests, and that data in the `Scope.Thread` scope means that it will not be shared between several threads.

The `@BenchmarkMode (Mode.AverageTime)` annotation says that what we want to measure is the average time required to execute our test, which, by default, measures throughput. The `@OutputTimeUnit (TimeUnit.MICROSECONDS)` annotation sets `timeunit`. We need to define it, as the default is in seconds, which is a very big scale for the benchmark.

The setup method is annotated with the `@Setup` annotation, which means that it does some preparation for the tests and it will be called to initialize the data for the test. It is similar to the `@Before` annotation from JUnit. Keep in mind that this method is executed only once before running the test in the fork on JVM. It is not executed before each test method is called. This means that the same test method will work with the same data after each iteration.

The actual test is in the method annotated with `@Benchmark` that executes the code that we are testing.

- Now that everything is set up, run a test and find out how fast our code is. First, let's build the project with our code and test it. To do this, go to the folder with the project and run the following command:

```
[user@localhost mybenchmark] mvn clean install
```

- Then, run the benchmark:

```
[user@localhost mybenchmark] java -jar target/benchmarks.jar  
--wi=10 --i=5 --f=1 --jvmArgs=-server
```

Here `wi` defines the number of warmup iterations, `i` the number of test run iterations, `f` says how many JVM forks to use, and `jvmArgs` are the parameters for forked JVM.

The output for each of our test methods should look like this:

```
# VM invoker: C:\Usres\User\jdk1.7.0\jre\bin\java.exe  
# VM options: -server  
# Warmup: 10 iterations, 1 s each  
# Measurement: 5 iterations, 1 s each  
# Threads: 1 thread, will synchronize iterations  
# Benchmark mode: Average time, time/op  
# Benchmark: org.benchmark.MyBenchmark.testBinarySearch1  
  
# Run progress: 0.00% complete, ETA 00:00:15  
# Fork: 1 of 1  
# Warmup Iteration   1: 74.562 ns/op  
# Warmup Iteration   2: 75.657 ns/op  
# Warmup Iteration   3: 79.575 ns/op
```

```

# Warmup Iteration 4: 75.718 ns/op
# Warmup Iteration 5: 76.432 ns/op
# Warmup Iteration 6: 75.965 ns/op
# Warmup Iteration 7: 73.987 ns/op
# Warmup Iteration 8: 75.677 ns/op
# Warmup Iteration 9: 76.326 ns/op
# Warmup Iteration 10: 77.050 ns/op
Iteration 1: 77.027 ns/op
Iteration 2: 75.870 ns/op
Iteration 3: 77.674 ns/op
Iteration 4: 81.460 ns/op
Iteration 5: 73.858 ns/op

Result: 77.178 ±(99.9%) 10.778 ns/op [Average]
  Statistics: (min, avg, max) = (73.858, 77.178, 81.460), stdev =
2.799
  Confidence interval (99.9%): [66.400, 87.956]

# Run complete. Total time: 00:00:18

Benchmark                               Mode  Samples   Score   Score
error  Units                               Mode  Samples   Score   Score
o.b.MyBenchmark.testBinarySearch1      avgt         5  77.178
10.778 ns/op

```

The output shows the runs executed for each fork and the final result. Here we can see that, on average, our method takes 77.178 nanoseconds to run.

8. Now that we have the results, what to do with them? Generally, these results make sense only when they are compared with something else. Let's try to make some changes to the code and see whether it helps our implementation of binary search to work faster. We can try to remove recursion and see how it's going to work. Create another class with the name, `org.benchmark.BinarySearch2`, and put the following implementation there:

```

package org.benchmark;

public class BinarySearch2 {
    public static int search(long[] arr, long value) {
        return search(arr, value, 0, arr.length-1);
    }
    private static int search(long[] arr, long value, int start,
int end) {

```

```
        while (end >= start) {
            int imid = start + ((end - start) / 2);
            if(arr[imid] == value)
                return imid;
            else if (arr[imid] < value)
                start = imid + 1;
            else
                end = imid - 1;
        }
        return -1;
    }
}
```

This is the iterative implementation, which doesn't use recursive calls.

9. Now let's update the benchmark class so that we can compare recursive and iterative implementations:

```
package org.benchmark;

import org.openjdk.jmh.annotations.*;
import java.util.Arrays;
import java.util.concurrent.TimeUnit;

@State(value = Scope.Group)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
public class MyBenchmark {
    private long[] data = new long[500000];

    @Setup
    public void setup() {
        for (int i = 0; i != data.length; ++i) {
            data[i] = (long) (Math.random() * (Long.MAX_VALUE-1));
        }
        Arrays.sort(data);
    }

    @Benchmark
    @Group(value = "bsearch")
    public int testBinarySearch1() {
        return BinarySearch1.search(data, Long.MAX_VALUE);
    }

    @Benchmark
```

```

    @Group(value = "bsearch")
    public int testBinarySearch2() {
        return BinarySearch2.search(data, Long.MAX_VALUE);
    }
}

```

The difference between this benchmark and the previous version is that this uses the benchmarking groups that give us a simple comparison of our implementations. The `@State` annotation now has to have the `Group` scope, otherwise the tests will be using different data instances, which is not what we would want, as we want algorithms to work in exactly the same conditions.

10. Now, rebuild the project:

```
[user@localhost mybenchmark] mvn clean install
```

11. Then, run the test again:

```
[user@localhost mybenchmark] java -jar target/benchmarks.jar
--wi=10 --i=5 --f=1 --jvmArgs=-server
```

The output is going to be slightly different from the previous one, because of the use of groups. The main difference in which we are interested is going to be at the end of the report:

```
# Run complete. Total time: 00:00:18
```

Benchmark	Mode	Samples	Score
Score error Units			
o.b.MyBenchmark.bsearch	avgt	5	66.929
1.663 ns/op			
o.b.MyBenchmark.bsearch:testBinarySearch1	avgt	5	79.717
2.289 ns/op			
o.b.MyBenchmark.bsearch:testBinarySearch2	avgt	5	54.141
1.209 ns/op			

What we can see here is that, for this particular configuration (which includes machine spec, version of JDK, OS, and so on), iterative implementation, which is implemented by the method, `testBinarySearch2()`, is on average faster than the recursive one, implemented by `testBinarySearch1()` ($54.141 < 79.717$).

After going through this recipe, you have learned how to run microbenchmark tests, how to interpret results, and how to compare the performance of different implementations. Ensure that you microbenchmark each hard task properly and remember that the results can vary significantly on different machines, JVM versions, and so on.

There's more...

JMH is a flexible framework that provides flexibility to the way it can be used. For example, if someone wants to run tests via the main method, without using `benchmarks.jar`, this can be easily achieved. To do this, just add the following main method to `MyBenchmark` and run it:

```
public static void main(String[] args) throws RunnerException {
    Options opt = new OptionsBuilder()
        .include(".*" + MyBenchmark.class.getSimpleName() + ".*")
        .warmupIterations(3)
        .measurementIterations(3)
        .forks(2)
        .build();
    new Runner(opt).run();
}
```

This example will give the same result as running the following command:

```
[user@localhost mybenchmark] java -jar target/benchmarks.jar --wi=3 --i=3
--f=2
```

We also recommend that you download the source code (see the *Downloading source and compiling JHM* recipe). Have a look at the recipe and JavaDocs, as the JavaDocs are well written and explain a lot about the framework.

See also

Similar to `jcstress`, this framework is also built and maintained by Aleksey Shipilëv who has his own blog and also delivers lectures in various conferences. We recommend you visit his home page (<http://shipilev.net/>), watch his videos on YouTube (for example, <https://www.youtube.com/watch?v=4p4vL6EhzOk>), and read some of his papers.

Downloading the source and compiling JHM

Similar to all other OpenJDK tools and projects, there is an option to download the source of the JHM and build it yourself. This might be required if a framework requires customization and an extension fix. Luckily the process is very easy and straightforward.

Getting ready

This recipe requires a machine with an Internet connection that is capable of running Mercurial and Maven. Basically, the requirements are the same as that for compiling and running `jcstress` (see the *Building and running jcstress* recipe).

How to do it...

The following steps will lead you through the process of downloading the source code and building JHM:

1. To start, let's run the following command to download the source files:

```
hg clone http://hg.openjdk.java.net/code-tools/jmh/ jmh
```

This command will download the source files and put them into the `jmh` folder.

2. The next step is to build the source code. The build requires Maven to be installed on the machine. Change the current folder to `jmh` and run the following command:

```
mvn clean install -DskipTests
```

This command should generate an output similar to the following:

```
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 3:14.052s
[INFO] Finished at: Sat Aug 02 19:43:39 BST 2014
[INFO] Final Memory: 38M/176M
[INFO] -----
```

This means that the build was successful.

3. Now, the final step is to change your benchmark project to use the version of JHM that was just built. Assuming the project references of the JHM version are by properties, just change the version of the JHM dependency in your project to `1.0-SNAPSHOT`:

```
<properties>
...
<jmh.version>0.9.3</jmh.version>
</properties>
```

How it works...

When you run Maven with the `install` target, it will put the newly built version of the artefact in the local repository. In this case, the version is `1.0-SNAPSHOT`. When another project has dependency on that version, Maven will pick the version from the local repository and use it.

10

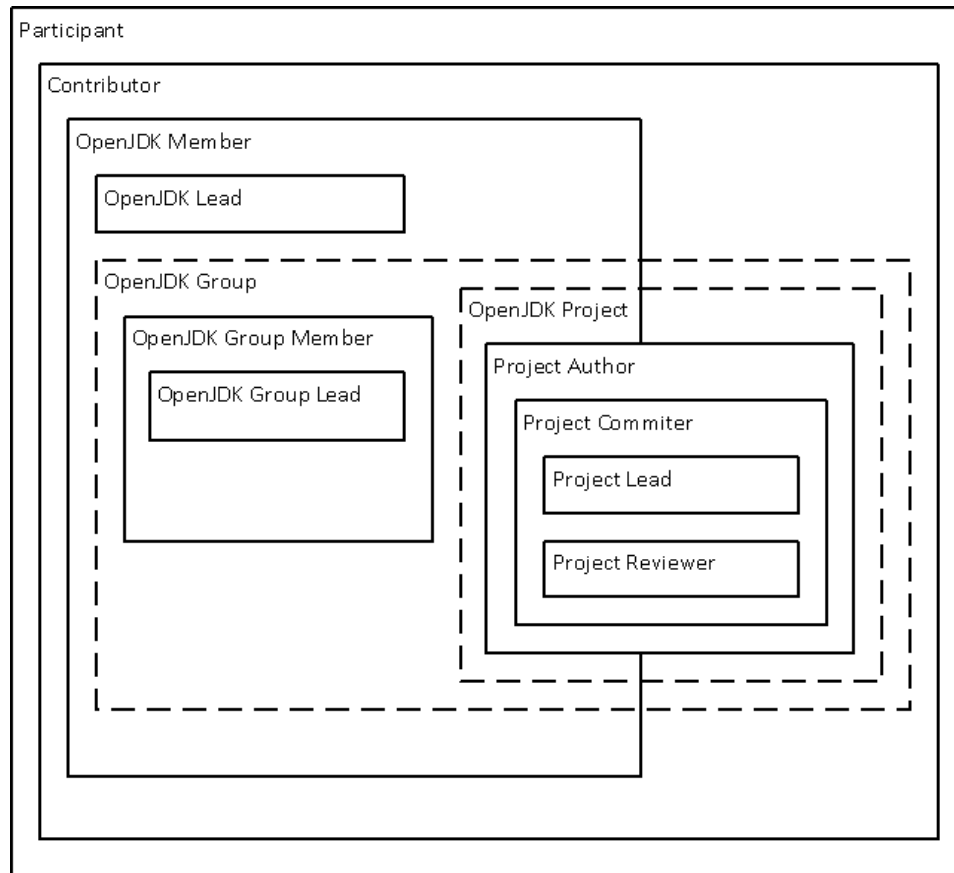
Contributing to OpenJDK

In this chapter, we will cover the following topics:

- ▶ Becoming a contributor
- ▶ Generating a patch with webrev
- ▶ Backporting OpenJDK v9 patches to OpenJDK v8
- ▶ Understanding OpenJDK groups
- ▶ Understanding OpenJDK projects
- ▶ Suggesting new JSRs
- ▶ Suggesting new JEPs

Introduction

The OpenJDK community consists of lots of people who have different roles and responsibilities in a project. Such a structure evolved as a consequence of the scale of the project and its significance, otherwise it wouldn't be controllable and you wouldn't be able to progress further. The way in which OpenJDK is managed and structured can be described in two hierarchies: one is functional and the other is governance. These two hierarchies intersect, but not much. The only role which is present in both is **JDK Lead**, which is an OpenJDK member appointed by Oracle to manage the Java release project.



Also, a contributor can become an OpenJDK member and participate in the OpenJDK groups. To become an OpenJDK member, the contributor needs to demonstrate a history of significant contribution and should be elected by the votes of the existing OpenJDK members. Being a member allows one to become an OpenJDK group member by joining one or more groups. Group members are nominated by existing group members so, to join a group, a member must have some history of contribution to that group. It is worth adding that the order of membership can be reversed—a contributor can be elected to become a group member and this can then become the route to be an OpenJDK member.

Each OpenJDK group is responsible for the area of JDK. For example, there is a *Swing Group*, *Security Group*, and so on. Each group has a group lead and group members. A group can sponsor projects, for example, as already mentioned, the Swing Group is a sponsor for projects such as *OpenJFX* and *Swing Application Framework*. As we mentioned earlier, any contributor can work on a sponsored project. The contributor is not required to be an OpenJDK member or a group member to do this.

The governance structure is represented by the *Governing Board*, which consists of several roles, as follows:

- ▶ **Chair:** This role is appointed by Oracle. This person is the lead of the Governing Board.
- ▶ **Vice Chair:** This role is appointed by IBM
- ▶ **OpenJDK Lead:** This role is appointed by Oracle. This person also leads the Java Release project.
- ▶ **Two At-Large members:** These two members are chosen by a vote of the OpenJDK members.

The governing board's responsibility is to define new processes or refine existing ones, and update and support the Bylaws. The Governing Board is empowered to resolve procedural disputes within community, but it is not an executive body. The latter means that it has no direct authority over the technical or release decisions. Interestingly, the Governing Board and also the OpenJDK group can also sponsor projects.

For those who are interested in the details of the hierarchy of OpenJDK roles, groups, projects, and their relationships, it is worth taking a look at the *bylaws* (<http://openjdk.java.net/bylaws>), which cover all these things in detail.

Becoming a contributor

Becoming a participant is very easy. The only thing that is required is to be interested in OpenJDK and, as a minimum, participate in discussions on mailing lists. Participants can ask OpenJDK members to add code into the code base, but they can't do it themselves. Contributors, however, have a more involved role. These people can submit patches and have more influence on the project.

Getting ready

To get ready to follow this recipe is easy; all that is required is a desire to become a participant. It is important to understand that it is not a quick process and can easily take several weeks or even months. So, be patient and do not give up.

As preparing a patch requires running the Kron shell script, a machine needs to have that shell installed. If a Windows machine is used, then Cygwin is required on the system.

How to do it...

The next steps will describe how to become a participant of OpenJDK. Keep in mind that this is not a deterministic process and, in practice, it might either need a few additional steps or some of the following steps might not be required:

1. The first step is to become visible to other members of the OpenJDK community. This can be achieved by participating in the discussions on mailing lists, and it would be good to suggest a couple of small fixes and ask one of the OpenJDK members or contributors to submit them. This will show that you are interested in OpenJDK and how to contribute to it.
2. Then, create an account in `Java.net` by filling the form at <https://java.net/people/new>. This account is needed for `Java.net`, OpenJDK bug tracking, and the source repository.
3. The next step is to sign the **Oracle Contribution Agreement (OCA)**. It needs to be downloaded from <http://www.oracle.com/technetwork/oca-405177.pdf>. Read the agreement carefully and understand it, before signing it. Luckily, it is just one page long. Then, sign it and e-mail the scan to `oracle-ca_us@oracle.com`. It might take a couple of weeks to receive a response. So, if the stipulated time has passed and you still haven't heard back, then it is worth sending a reminder.
4. Once you get back the OCA response from Oracle, it's time to find out what your contribution is going to be. Probably, the easiest thing is to find a bug and work on a fix for it. Bugs can be found on the OpenJDK JIRA at <https://bugs.openjdk.java.net/>. Pick something that is not assigned to anyone. Be sure to choose something that will not require interface changes or any significant development.

5. Before doing the actual work, it would be a good idea to discuss the bug and the proposed fix on the mailing list for the appropriate project. It is recommended to use the format, `<bugid>: <bug_description>` as the subject, for example, `JDK-8042253: StressRedefine tests timeout`.
6. Carry out a fix. It is good idea to do the `jtreg` regression test for any work that was done in the scope of the fix.
7. Generate a patch for review using `webrev`: `[user@localhost hotspot]$ ksh ./make/scripts/webrev.ksh`. Make sure that the generated `webrev.zip` file is available for the relevant community. For example, it can be put on a file hosting with public access. Regular contributors prefer to use `cr.openjdk.java.net` for this purpose. For more details about `webrev`, see the *Generating a patch with webrev* recipe.
8. To post a patch, use the format `RFR <bugid>: <bug_description>`, as the subject for the mailing list. For example, `RFR JDK-8042253 StressRedefine tests timeout`, where **RFR** refers to **Request for Review**. Give the description of the patch and provide a link to the `webrev` file generated in the previous step.
9. A review always means that there will be some cleanup, code changes, more tests, and so on. Carry out the updates, update `webrev`, and notify other members that changes have been made. Be ready since the process might need several iterations.
10. Once everyone agrees that the fix is of a good quality, the sponsor will push the fix into the repository. At this stage, you might be asked to produce a changeset. Note that this step can take several weeks to execute.

See also

- Some additional information on the code review process is available at <http://openjdk.java.net/guide/codeReview.html>. There is also documentation available on the change process as a whole. It is worth having a look at it at <http://openjdk.java.net/guide/changePlanning.html>.

Generating a patch with webrev

Doing any update in a software product normally requires some kind of review process. The code should not be submitted into a repository unless someone else has already looked at it. In OpenJDK, the tool for this purpose is `webrev`, which allows you to create code reviews that can be shared with other members of the community.

Getting ready

Webrev is a Korn shell script, which means that Korn shell must be installed on the machine before you can use it. On a Linux machine, run `yum install ksh` or an equivalent command. On a Windows machine, ensure that `ksh` is included in the Cygwin installation.

Apart from the environment set up to create the patch review, some changes must be made in the source code as well. As an easy example, we can just modify some comments in a file, as discussed next. Open the file, `hotspot/src/os/linux/vm/jvm_linux.cpp` and update the header or just add a comment.

How to do it...

As webdev is a shell script, start by launching your shell and follow the given steps. It is assumed that the current folder is `jdk` source root and the amended file is `hotspot/src/os/linux/vm/jvm_linux.cpp`:

1. First, change the current folder to `hotspot`; it's not strictly required, but it will make the webrev script do less work:

```
cd ./hotspot
```

2. Run the webrev script:

```
[user@localhost hotspot]$ ksh ../make/scripts/webrev.ksh
SCM detected: mercurial
No outgoing, perhaps you haven't committed.
Workspace: /home/user/openjdk/jdk7u/hotspot
Compare against:
http://hg.openjdk.java.net/jdk7u/jdk7u/hotspot
Output to: /home/user/openjdk/jdk7u_clean/hotspot/webrev
Output Files:
src/os/linux/vm/jvm_linux.cpp
patch cdiffs udiffs sdiffs frames old new
index.html: grep: /home/user/.hgrc:
No such file or directory
Done.
Output to: /home/user/openjdk/jdk7u/hotspot/webrev
```

As can be seen from the output, it detected that `jvm_linux.cpp` was changed and generated review files which were put in the `hotspot/webrev` folder.

3. Now, in the `hotspot` folder, locate the `webrev.zip` file and make it available to anyone who is interested in the changes that you have just made. This file will have the same content as the `webrev` folder.

How it works...

As you have already noticed, webrev is just a shell script and, if you need to see how it works, then it should be relatively easy to do so. All it does is scan your disk for changes and compare them with the parent Mercurial forest. Then, based on the comparison, it generates various reports and a patch file. Finally, all the files are packed into a ZIP file, which is easy to share with other community members or with anyone who just wants to have a look at the change.

There's more...

Webrev has some command options which are displayed if you enter a command, which it doesn't understand. For example, try to run this:

```
[stas@localhost hotspot]$ ksh ../make/scripts/webrev.ksh ?
```

This will print all the available command-line options and environment variables that can affect the execution of the script.

See also

- ▶ More information and several examples are available on the `openjdk.java.net` website at <http://openjdk.java.net/guide/webrevHelp.html>
- ▶ The overall OpenJDK code review process is described at <http://openjdk.java.net/guide/codeReview.html>

Backporting OpenJDK v9 patches to OpenJDK v8

In the scope of one of the initiatives for the improvement of the source code structure (<https://bugs.openjdk.java.net/browse/JDK-8051619>) there are some significant changes in the way how the source files are located in OpenJDK v9. It means that if someone is making a patch for OpenJDK v9 and wants these changes to be applied to OpenJDK v8, he/she has to follow a special procedure. That procedure will do the required transformation in the path of changed files.

Getting ready

We need a computer which can run bash shell, that is, a Linux or Windows computer with Cygwin and the source code of OpenJDK v9 and OpenJDK v8.

How to do it...

Following these steps, a developer can learn how to port changes from OpenJDK v9 to OpenJDK v8:

1. There is a special script that was created to help to port patches from OpenJDK v9 to OpenJDK v8. This script is located in `common/bin/unshuffle_patch.sh`. Run this script with the `--help` argument to see its usage:

```
[user@localhost jdk9]$ common/bin/unshuffle_patch.sh --help
Usage: common/bin/unshuffle_patch.sh [-h|--help] [-v|--verbose]
<repo> <input_patch> <output_patch>

where:
<repo>          is one of: corba, jaxp, jaxws, jdk, langtools,
nashorn

[Note: patches from other repos do not need
updating]
<input_patch>   is the input patch file, that needs shuffling/
unshuffling
<output_patch>  is the updated patch file
```

If you can see the help output, it means that the script is available and should just work fine.

2. Now, just make the required change in the source tree, commit the code, and generate a patch. For our example, we will edit the `Sockets.java` file. Just add a new line with a few comments in it as shown:

```
[user@localhost jdk9]$ vi ../jdk/src/java.base/share/classes/jdk/
net/Sockets.java
now commit the change:
[user@localhost jdk9]$ cd jdk
[user@localhost jdk]$ hg commit
```

3. Next, get the revision number of the changeset:

```
[user@localhost jdk]$ hg log -l 1
changeset: 11063:9742f66b011
tag:       tip
user:      User <user@user.org>
date:      Sun Dec 14 21:16:27 2014 +0000
summary:   test commit
```

4. Now, export the changeset that we just created to a patch file in an extended GIT format:


```
[user@localhost jdk]$ hg export -r 11063 --git > 11064.patch
```
5. Then, run the script to make the patch compatible with the OpenJDK 8 source code tree:


```
[user@localhost jdk]$ ../common/bin/unshuffled_patch.sh jdk 11063.patch 11063_updated.patch
```
6. Finally, change the current folder to the jdk directory in the jdk8u source root, copy the updated patch to it, and apply it:


```
[user@localhost jdk]$ cp ../../jdk9/jdk9/11063_updated.patch ./
[user@localhost jdk]$ hg import 11063_updated.patch
```

And that's it. Now all that is required is to commit the changes and then follow the normal process of making the changes in OpenJDK.

See also

- To get some more information and to be up-to-date with the changes in the utilities, it is recommended that you visit its home page at <http://cr.openjdk.java.net/~chegar/docs/portingScript.html>

Understanding OpenJDK groups

OpenJDK groups look at wide areas of OpenJDK and define what projects are required to support these areas. For example, the Compiler Group's sponsored project Coin added new language features in JDK7, and the HotSpot Group's sponsored project Graal makes VM functionality available via a set of APIs. The groups usually have a much longer life than the projects and do not appear and disappear very often.

Getting ready

All that is required is a computer with access to the Internet, a web browser, and some patience.

How to do it...

The following steps demonstrate documentation and some examples of groups on OpenJDK. They'll also show what groups are, how they are managed, created, and what they do:

1. To become familiar with the definition of groups in OpenJDK, go to <http://openjdk.java.net/groups/>. There is information about various processes and procedures required to support groups and to make them function well. These procedures include the following topics:
 - ❑ Proposing a new group.
 - ❑ Nominating a contributor to become a group member.
 - ❑ Nominating a group member to become an OpenJDK member
 - ❑ Nominating a group lead
2. Each group has some web content that is available publically. It can be found on the left-hand side of <http://openjdk.java.net/>, under **Groups**. Usually the content on a group's page has a relatively standard structure and contains information such as the introduction, a list of specifications supported by the group, documentation, some guidelines on how to contribute, where to find the source code, and so on. Also, there are links to the mailing lists, blogs, and contact details. As an example, take a look at the JMX Group webpage at <http://openjdk.java.net/groups/jmx/>.
3. We also recommended that you have a look at the list of projects and see which groups they belong to. This will give some idea about the relationship between groups and projects. The list of projects can be found on the left-hand side of <http://openjdk.java.net/> under **Groups**.

See also

It is worth spending some time exploring <http://openjdk.java.net/> to see what is available and which groups exist. As we have already mentioned, group creation doesn't happen very often and no new groups have been created since 2007. The last one was the Conformance Group, which was proposed in September 2007. You can find the proposal e-mail in the mailing list archive at <http://mail.openjdk.java.net/pipermail/announce/2007-September.txt>. See the last message in that file.

Understanding OpenJDK projects

OpenJDK projects aim to deliver some kind of an artefact, which can be a source code, documentation, or something else. Projects are sponsored by one or more OpenJDK groups. By their nature, projects are much more short-lived than groups and often can cover the implementation of JEPs or JSRs.

How to do it...

This recipe takes you through some sources of information that are available about OpenJDK projects and gives some high-level information about them:

1. Similar to OpenJDK Groups, there is a web page that gives some definitions about the projects and defines how they function. This page is available at <http://openjdk.java.net/projects/>. Among the list of procedures, you will find the following:
 - ❑ Becoming an Author
 - ❑ Appointing a Contributor to be an Author
 - ❑ Nominating a Contributor or Author to be a Committer
 - ❑ Nominating a Committer to be a Reviewer
 - ❑ Proposing a New Project
2. Very similar to groups, each project also has its own page on <http://openjdk.java.net/>. The list of projects and the links to their pages can be found on the left-hand side of the website under **Projects**. The project page is not very informative, but might have a link to a wiki, which usually contains lots of information.
3. As an example of how projects are created, we can look at the proposal for the project *Sumatra*, which can be found at <http://mail.openjdk.java.net/pipermail/announce/2012-September/000135.html>. This thread also contains the voting results and, therefore, the decision to create that project.

See also

- As usual, it is recommended that you spend some time exploring <http://openjdk.java.net/> to see which projects are available.

Suggesting new JSRs

A **Java Specification Request (JSR)** is the request for a change to the specifications of the Java language, APS, JVM, and so on. Such changes are governed by the **Java Community Process (JCP)**, where every member of the community can register and participate in the review.

This recipe is written as if you were submitting a JSR, but keep in mind that JSRs are not usually submitted by a single individual, but by a group of experts who come out with a proposal. The group has a Spec Lead who submits the JSR. So, this recipe is more about giving you a high-level understanding of the process, helping you to see how JSRs work, and what it is all about. To get some deeper insight, see the resources provided in the *See also* section.

In the following recipe, it can be seen that being a spec lead for JSR is not only a technical position, but it also involves doing lots of work with people and requires one to possess a good amount of soft and leadership skills. The leader has to push the project forward and has to be capable of handling difficult situations. There can be cases when some members of group of experts, for some reason, can't participate any more, or there is clash of personalities. The other possible situation is when the JSR itself faces complicated questions and is challenged by the other community members. Such questions have to be answered clearly and with enough knowledge and passion for people to believe that it's worth going forward and including that JSR into the JDK.

Getting ready

As this is a more procedural recipe, all that is required is a computer with access to the Internet and a web browser.

How to do it...

The following steps, on a high-level, go through the stages of a JSR lifecycle, starting from the idea and finishing with the actual implemented change:

1. As a start, it would be good to create a JCP account. This is required to submit a JSR proposal and to participate in any part of a JCP. The process is very simple. Just go to <https://jcp.org/en/home/index> and follow the steps in the registration link.
2. It is good idea to explore the site and see what's already available there. The list of all JSRs is available at <https://www.jcp.org/en/jsr/all>. Since the whole list can have too much information, there are options to filter JSRs by their stage in the approval process (<https://www.jcp.org/en/jsr/stage>), by technology (<https://www.jcp.org/en/jsr/tech>), by committee (<https://www.jcp.org/en/jsr/ec>), or by the platform (<https://www.jcp.org/en/jsr/platform>). There is also a list of JSRs by the ballot results, where you can find the results of voting for each year at https://www.jcp.org/en/jsr/vote_summary.

3. If your proposal is something worth doing, then, similar to all OpenJDK change processes, describe the proposal on the relevant mailing list. This will ensure that the whole thing makes sense and helps to improve the quality of the material. As a spec lead, you need to have followers, that is, a group of experts who will participate in the JCR and push it forward. Such people can be found on mailing lists, relevant forums, or can be anyone who has the right mindset to participate and feels that your idea is worth doing.
4. To make an actual submission, fill the **Java Specification Request Proposal**, which is available at <https://jcp.org/en/jsr/proposal>. After submission, the JSR has to go through the **JSR Approval Ballot**, which will determine whether the initial JSR submission should be approved.
5. In the next stage, the group has to start working as a team, push the proposal forward, discuss it, and answer questions from other members of the community. It would be a good idea to have regular team calls, and regular face-to-face discussions can also be beneficial. This stage is probably the most important one, as it forms the precise shape of the JSR.
6. All discussions have to be publically available and the expert group that has proposed the JSR has to publically answer all the raised questions. This also means that there has to be a publically available archive of all the communication on the matter.
7. When all the responses for all the comments and questions have been received and the JSR is updated accordingly, it is ready for the final submission. The spec lead is responsible for completion and submitting the **Technology Compatibly Kit (TCK)** and the **Reference Implementation (RI)**. If a JSR is targeted to several environments, then it might be required to submit a TCK and RI for each platform. The submission has to follow the process described in the *Final Release* section of the JCP 2 process at <https://jcp.org/en/procedures/jcp2#4>.
8. After the final successful approval ballot, the spec lead will become the maintenance lead and the JSR will move to the maintenance stage. For details, see the *Maintenance* section of JCP 2 process at <https://jcp.org/en/procedures/jcp2#5>.

See also

As the whole process is governed by JCP, it would be a good idea for you to go through the document at <https://jcp.org/en/procedures/jcp2>, which describes the latest version of the process. At the time of writing, the latest version was 2.9.

On the JCP website at https://jcp.org/en/resources/speclead_life, there is a good article about being a spec lead that has some good insights and advice. It covers almost all the aspects of the role, starting from the submission of a JSR, and ending with building a team and communicating with people. It is definitely worth reading.

As part of the JSR process, one is required to submit the **Technology Compatibility Kit (TCK)**. This is the suite of tests that are designed to ensure that a particular implementation is compliant with the JSR. This piece can be considered as one of the most complicated parts of the JSR application. The most common tool to implement the set of tests is JT Harness, but there are also some cases where the TCK might be implemented based on JUnit or TestNG.

The next revision of JCP is going to update some of the current process. This revision is covered by JSR 358. To get more information about it, visit its home page at <https://java.net/projects/jsr358/pages/Home> and the JRS page at <https://jcp.org/en/jsr/detail?id=358>.

Suggesting new JEPs

JEP is the abbreviation for JDK Enhancement Proposal. It means a relatively big change in OpenJDK that requires significant implementation effort, but it doesn't imply changes in the Java specification. The definition of JEP is covered in *JEP 1 JDK Enhancement-Proposal & Roadmap Process* which explains the details of definition of JEP, the process and the required documentation. As defined in JEP 1 (<http://openjdk.java.net/jeps/1>), JEP has to meet at least one of the following criteria:

- ▶ It requires two or more weeks of engineering effort
- ▶ It makes a significant change to the JDK, or to the processes and the infrastructure by which it is developed
- ▶ It is in high demand by the developers or the customers

This recipe will cover the definition of JEPs, their lifecycle, and how to find the latest information about them. It will also go through steps that are required for the creation of a JEP. Such an approach will allow you to get a good understanding of the process and see what JEPs are.

Getting ready

This recipe doesn't require any special tools. All you need is this book and, preferably, a computer with a web browser.

How to do it...

To start with, it is a good idea to have a look at what has already been submitted as JEPs. To get the full list of JEPs, just go to <http://openjdk.java.net/jeps/>. There, you can find the complete list of JEPs with their status, name, and some more information about them. Here is an example of what you can see there:

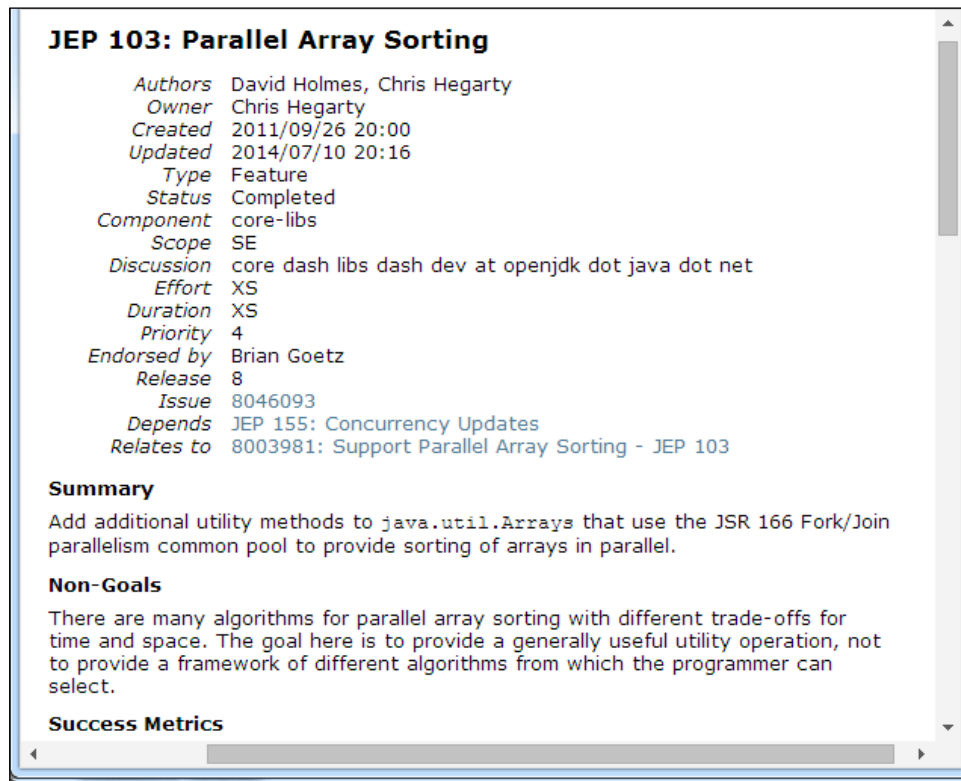
JEP 0: JEP Index

<i>Owner</i>	Mark Reinhold
<i>Created</i>	2011/08/24 17:33
<i>Updated</i>	2014/07/11 20:41
<i>Type</i>	Informational
<i>Status</i>	Active

This JEP is the index of all JDK Enhancement Proposals, known as JEPs.
See JEP 1 for an overview of the JEP Process.

P Can		1	JDK Enhancement-Proposal & Roadmap Process
P Can		2	JEP Template
F Com 8	spec/lang	101	Generalized Target-Type Inference
F Can 9	core/lang	102	Process API Updates
F Com 8	core/—	103	Parallel Array Sorting
F Com 8	—/—	104	Annotations on Java Types
F Com 8	spec/lang	105	DocTree API
F Com 8	spec/lang	106	Add Javadoc to javax.tools
F Clo 8	core/—	107	Bulk Data Operations for Collections
F Clo	core/—	108	Collections Enhancements from Third-Party Libraries
F Com 8	core/—	109	Enhance Core Libraries with Lambda
F Sub 9	core/net	110	HTTP 2 Client
F Can	core/—	111	Additional Unicode Constructs for Regular Expressions
F Com 8	core/—	112	Charset Implementation Improvements
F Com 8	security/—	113	MS-SFU Kerberos 5 Extensions
F Com 8	security/—	114	TLS Server Name Indication (SNI) Extension
F Com 8	security/—	115	AEAD CipherSuites
F Can	security/—	116	Extended Validation SSL Certificates
F Com 8	spec/lang	117	Remove the Annotation-Processing Tool (apt)

As you can see from the list, the table of JEPs has several columns, which give a brief overview and some additional information. For example, the first column is the type of the JEP, P stands for Process, F for Feature, and so on. It is not that hard to find out the meaning of specific values if you click on **JEP** and have a look at its header:



JEP 103: Parallel Array Sorting

Authors	David Holmes, Chris Hegarty
Owner	Chris Hegarty
Created	2011/09/26 20:00
Updated	2014/07/10 20:16
Type	Feature
Status	Completed
Component	core-libs
Scope	SE
Discussion	core dash libs dash dev at openjdk dot java dot net
Effort	XS
Duration	XS
Priority	4
Endorsed by	Brian Goetz
Release	8
Issue	8046093
Depends	JEP 155: Concurrency Updates
Relates to	8003981: Support Parallel Array Sorting - JEP 103

Summary

Add additional utility methods to `java.util.Arrays` that use the JSR 166 Fork/Join parallelism common pool to provide sorting of arrays in parallel.

Non-Goals

There are many algorithms for parallel array sorting with different trade-offs for time and space. The goal here is to provide a generally useful utility operation, not to provide a framework of different algorithms from which the programmer can select.

Success Metrics

Assuming that we have something completely new that is not in the list, and which definitely has to become a part of OpenJDK:

1. The next step is to read through *JEP 1: JDK Enhancement-Proposal & Roadmap Process*, which is available at <http://openjdk.java.net/jeps/1>. This covers the process and some mechanics of the process.
2. The next step is to do some more reading. *JEP 2: JEP Template* contains the template of the JEP. This template has to be filled in with the proposal details. The details will cover the overview, testing, dependencies, and so on. It is worthwhile looking at other JEP examples to fill the template. There is also a sample draft, available at <http://cr.openjdk.java.net/~mr/jep/draft-mcimadamore-inference-01.md>.

3. Before attempting to publish a JEP, it would be a good idea to submit the proposal to the appropriate OpenJDK mailing lists for discussion. This will help you to produce a high quality proposal.
4. When the proposal is ready for submission, send it to `jep-submit@openjdk.java.net`. After this, assuming that the proposal has adequate quality, it will be available in the repository (<http://hg.openjdk.java.net/jep/jeps>) and the webpage which was mentioned in step 1 of this recipe.
5. After this, there will be several rounds of updates and discussions, which might eventually result in the JEP being approved and having its status changed to `Candidate`. This means that it is accepted for inclusion and has a good chance of being funded for one of the next versions. It would be worth mentioning that a JEP can also be rejected, which means that it was decided that it was not worth implementing at all.
6. When a JEP is transferred to the status `Funded`, it means that the group or area lead is happy to fund it. This means that the actual implementation can now be initiated and will be included in one of the future versions.

How it works...

As was already mentioned earlier, the detailed process is described in *JEP 1: JDK Enhancement-Proposal & Roadmap Process*. Similar to all OpenJDK changes, this process requires lots of involvement from the community and OpenJDK members.

See also

- ▶ *JEP 1: JDK Enhancement-Proposal & Roadmap Process* at <http://openjdk.java.net/jeps/1>
- ▶ *JEP 2: JEP Template* at <http://openjdk.java.net/jeps/2>
- ▶ It would be good for you to take a look at some exciting discussions of JEP on mailing lists at <http://mail.openjdk.java.net/pipermail/jdk9-dev/2014-May/000781.html>.
- ▶ There is a JEP at <http://cr.openjdk.java.net/~mr/jep/jep-2.0-02.html> that suggests some changes in the existing process. The current state is *Draft*, but eventually most of the proposals will be applied to the current process.

11

Troubleshooting

In this chapter, we will cover the following topics:

- ▶ Navigating through the process workflow
- ▶ Submitting a defect to the OpenJDK bug systems
- ▶ Creating a patch using NetBeans
- ▶ Creating a code review

Introduction

OpenJDK's openness is its most valuable feature in our fast-paced world. We can be sure that it will never vanish and be forgotten when it is needed. If its support is discontinued, we can support it ourselves. We could fix bugs independently, since the code is open.

However, OpenJDK, as well as the Oracle Java implementation, still contains a lot of issues. Some of them are security issues and should be fixed as soon as possible. Some of the issues may be almost invisible or even valuable only for a virtually nonexistent percent of customers. They might never be fixed in production JDK, but everyone has an opportunity to try to do it for himself/herself.

OpenJDK in the past used a Bugzilla bug tracker to track bugs. Bugzilla is a well-known, but morally outdated project, initially written in 1998 by Terry Weissman. It was open source from the beginning, and it is used even now by thousands of people around the world. It is very simple and easy to use.

However, some time ago, the OpenJDK foundation decided to switch from Bugzilla to JIRA, a proprietary but powerful bug tracking system mostly written in Java. JIRA provides support for different types of workflow, such as Scrum, Kanban, and custom Agile workflows, as well as incorporating all the Bugzilla features.

Sometimes there are opportunities to enlarge the OpenJDK functionality and create a fully-functioning JSR prototype to contribute to OpenJDK development with great benefits to your projects. Sometimes you need to switch to other implementations, such as GNU classpath, or even merge some solutions into your specific version.

In this chapter, we will see how the process workflow is organized, what steps are needed to be followed, and how to cope with the community process. Also, we will see how to submit a bug, and create a patch on it.

Navigating through the process workflow

The knowledge of process workflows, and of how things are done in complex projects like OpenJDK, is critical for a developer. In this recipe, we will explain how things are done while contributing to OpenJDK. We will explain how the work is organized, and how the team co-operates.


Getting ready

We will need an Internet connection and a browser.

How to do it...

We will go through the OpenJDK process workflow and see what to do to contribute to OpenJDK. There are a few initial steps to be followed before you start with the workflow:

1. Firstly, you need to become a contributor. To do so, you need to sign an Oracle Contributor Agreement, and e-mail it to Oracle.



You may find these links useful:

How to become a contributor at <http://openjdk.java.net/contribute/>

Know what is discussed at <http://mail.openjdk.java.net/mailman/listinfo>


2. Then you need to find something interesting to work with.
3. Now, you are free to discuss and submit patches using the JIRA bug tracking system.

Let's suppose that you've found a bug:

1. When the bug is found, check whether it's already in JIRA.
2. Click on **Issues** and search for issues. You will see the following screen:

T	Key	Summary	Assignee	Reporter	P	Status	Resolution	Created	Updated
	JDK-8042906	Remove use of reflection in ORB	Sean Coffey	Sean Coffey	4	In Progress	Unresolved	2014-05-11	2014-05-11
	JDK-8042905	Submit a patch set to Google's octane suite that does better verification	Marcus Lagergren	Marcus Lagergren	4	New	Unresolved	2014-05-11	2014-05-11
	JDK-8042904	apple.security.getSalt() calling generateSeed()	Unassigned	Sean Coffey	4	New	Unresolved	2014-05-11	2014-05-11
	JDK-8042903	PKCS12 KeyStore should not call secureRandom.generateSeed()	Sean Coffey	Andreas Sterbenz	3	Open	Unresolved	2014-05-11	2014-05-11
	JDK-8042902	Test java/net/net6Address/serialize/net6AddressSerializationTest.java fails intermittently	Unassigned	Eric Wang	3	New	Unresolved	2014-05-10	2014-05-10
	JDK-8042901	Allow com.sun.management to be in a different module to java.lang.management	Unassigned	Alan Bateman	3	New	Unresolved	2014-05-10	2014-05-10

3. In the highlighted section, you can see a filter panel. Use it in combination with the keyword search. Also, notice the **Switch to Advanced** link in the right corner.
4. If the bug is not present already in JIRA, create a JIRA issue.

[ JIRA is a commonly used bug tracker tool. You can find how to add issues at <https://confluence.atlassian.com/display/JIRA/>.]

5. If you don't know how to resolve the issue, this is the successful ending of your involvement here. The OpenJDK team will be grateful to know all usable information about this problem. If you have anything to add, you are welcome to post information in comments.
6. However, if you feel that you can resolve the issue, comment on it and indicate that you are working on the solution. In some circumstances, you need to wait until the JIRA ticket is assigned to you. Individual workflows use different approaches in this regards.
7. Then, resolve the solution and create the patch.

The patch creating process is described in the *Creating a patch using NetBeans* recipe in this chapter. you will also find some useful information about building, debugging, and editing OpenJDK code in *Chapter 8, Hacking OpenJDK*.

8. If you are a committer (for example, you have the commit right to the OpenJDK repository), you are free to create a code review using webrev (use information from <https://bitbucket.org/adoptopenjdk/betterrev> to learn how to create a code review). When it is approved, you may commit your changes to the repository and mark your issue as resolved.
9. If you are not a committer, you can also create a webrev review, but the commit process differs slightly. You might push your code changes to the Bitbucket repository along with a pull request. It is described in detail in the *Creating a code review* recipe. Read more about the OpenJDK testing process in *Chapter 9, Testing OpenJDK*.

How it works...

OpenJDK has an established workflow, which can differ from one project to another. In all cases, though there are rules of teamwork, one should follow them in order to make the work productive.

Under the hood of this process, there are other team members whose function is to review and test the changes.

Submitting a defect to the OpenJDK bug systems

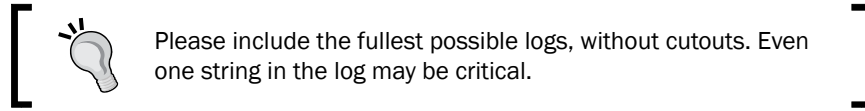
This recipe will show you how to submit a defect to OpenJDK bug systems. This bug system is used by people who do not have developer access to OpenJDK. For OpenJDK developers, there is JIRA.

How to do it...

We will consider the necessary steps to submit a defect to OpenJDK.

First, we will describe some prerequisites to fill a bug report:

1. First, do a quick search for bugs. It is possible that a similar or even identical bug has already been created. Even if you have something more to say on the subject, please do it in the existing topic without creating a new one.
2. Then, think about the defect's reproducibility. There is a need to describe in detail how to reproduce your defect and cases in which it can and cannot be reproduced.
3. Also, you can make and add logs, traces, and other debug information.



4. Also, you may include screenshots along with your logs, especially when you are describing an UI issue.
5. Try to include as much system information as possible. For example, for the *Sumatra* project, even your graphics card driver version may be helpful.

If we need to report an issue without referring to code pieces to describe it generally and not specifically, we can fill the standard bug report as shown here:

1. Fill the fields as shown in the following screenshot:

** indicates required fields*

*** Type:** *Bug: A behavior that violates the specification.
RFE: A feature that should be added to the product.*

*** Product/Category:**

*** Subcategory:**

*** Release:**

*** Operating System:**

2. Then click on **Continue**.

You will then be required to provide some information to receive the feedback from the team:

User Info
Please give us some information about yourself. Be sure to include a valid email address. You will receive confirmation and subsequent updates regarding your report via email.

*** Your Name:**

*** Company:**

*** Email:**

In addition, Oracle respects your desire for privacy. Personal data collected from this program will not be sold, given or shared with organizations external to Oracle. We will use this data for communications with you to clarify issues regarding the report you submitted and/or status of that report. The issues that you report may be made publicly available, however your personal data will be kept confidential. If you are not comfortable with the above conditions, please do not press the Submit button. If you have any questions, please refer to our [Privacy Policy](#).

How it works...

The OpenJDK project has a bug report system for non-developers to which you can submit your changes where it's convenient. However, it has a JIRA bug tracker, which is used by developers to work in more complex ways with various bugs.

Creating a patch using NetBeans

This section will show you how to create a patch of the OpenJDK project using NetBeans.

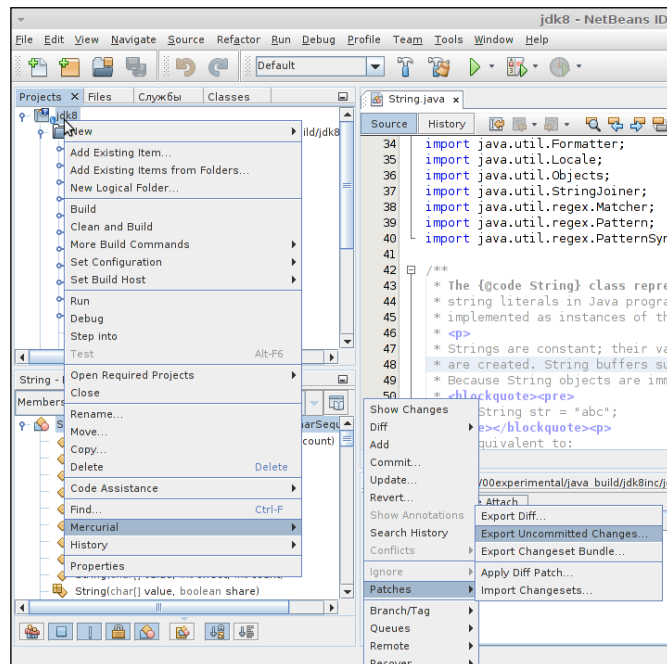
Getting ready

You will need a configured development environment. It is preferable that you use NetBeans since NetBeans is a standard tool for developing and debugging OpenJDK.

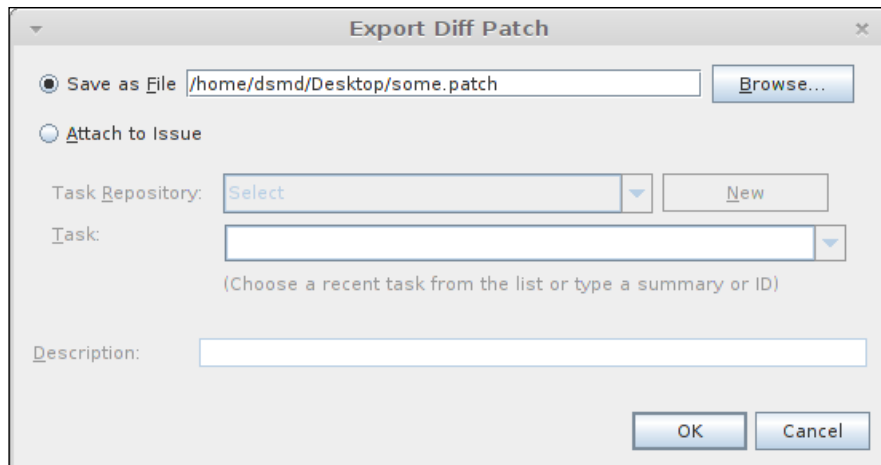
How to do it...

We will do a simple patch creation with Mercurial and NetBeans by following these steps:

1. Open the OpenJDK project with NetBeans.
2. Make some changes in the project code as shown:



3. Then, follow the chain of context menus, as shown in the preceding screenshot.
4. Click on the **OK** button.



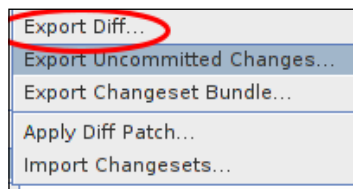
5. Your patch will be saved to the specified directory. Later, you can apply patches using the **Apply Diff Patch** item in the Mercurial context menu.

How it works

NetBeans calls a program that exports a diff (your uncommitted changes versus the head revision in the repository) to a file that can be read and applied when needed.

See also...

Also, diffs between different divisions can also be exported. To do that, simply click on the highlighted item:



Creating a code review

OpenJDK uses webrev as the code review tool. The process of creating a code review and handling lies on the shoulders of the AdoptOpenJDK community, which developed a web tool called **Betterrev**. Review Board along with the Mercurial plugin can be used as well. It automatically generates code reviews, synchronizes with Oracle repository, and performs other useful tasks.

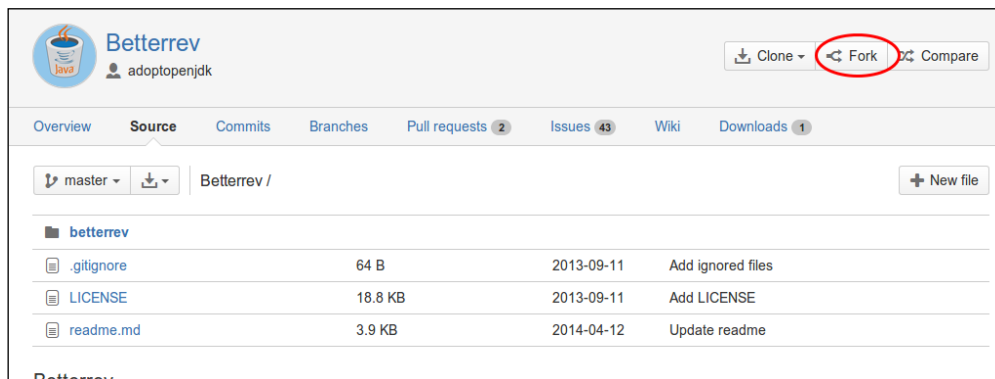
Getting ready

We will need a computer capable of building OpenJDK and which can handle a large amount of code. Also, we will need a development environment installed, as described in *Chapter 8, Hacking OpenJDK*. We will also need Internet access.

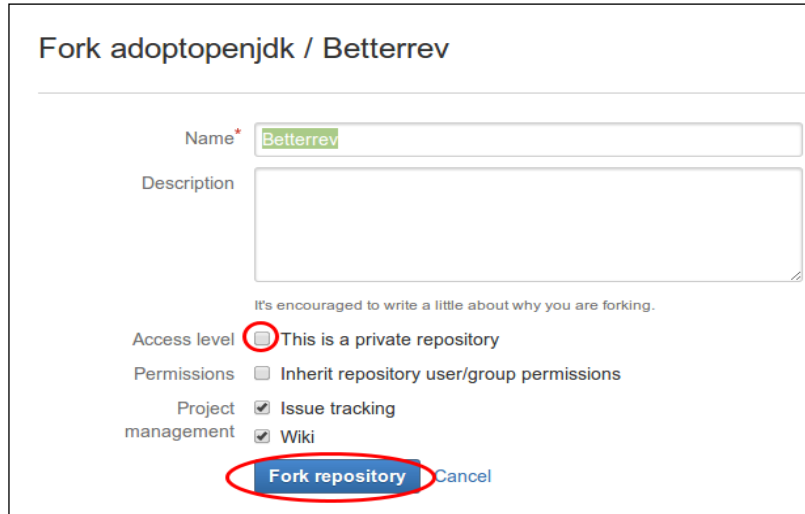
How to do it...

We will create a code review using the Betterrev tool, by following the given steps:

1. First, let's clone a repository from Bitbucket.
2. Go to the Betterrev Bitbucket at <https://bitbucket.org/adoptopenjdk/betterrev/src>:



3. Fork it, using the highlighted button:



Fork adoptopenjdk / Betterrev

Name*

Description

It's encouraged to write a little about why you are forking.

Access level ☒ This is a private repository

Permissions ☐ Inherit repository user/group permissions

Project management ☒ Issue tracking ☒ Wiki

4. Specify the repository name in the **Name** field, and add a description if needed.
5. Check the **This is a private repository** checkbox if you want to make your repository private.
6. Click on the **Fork repository** button.
7. You will see the following screen for a while. It will take some time, but don't worry, all is going well.



Cloning repository

Please wait a moment while we clone your repository.

8. Make changes that has to be reviewed in the code.
9. Commit them to your local repository.

10. Perform a pull request as shown:

Create a pull request

amironchenko / Betterrev
Created 2 minutes ago, updated 2 minutes ago
master

→

adoptopenjdk/betterrev
master

Title* Fix of the most important issue ever

Description
blah blah

Reviewers
Alex Kaska

Close branch ☐ Close master after the pull request is merged

Create pull request

Betterrev will automatically generate a review for your issue.

12

Working with Future Technologies

In this chapter, we will cover the following topics:

- ▶ Building OpenJDK 9 on Mac OS X using Clang
- ▶ Building OpenJDK 9 on Windows using MSYS
- ▶ Running and testing the early access preview of OpenJDK 9
- ▶ Using Jigsaw
- ▶ Building OpenJDK 9 with Graal
- ▶ Building OpenJDK 9 with Sumatra

Introduction

Java is often criticized because of some degree of conservatism, where major language changes are concerned. However, the recent Java 8 release has done a lot to relieve the worry that Java will remain conservative and frozen in time.

However, there are more changes coming. Java 9 is believed to support some long-awaited features that will possibly take it to a completely new market and level of programming.

In the past, annotations support and generics have caused a revolution in Java programming. The way of thinking was changed, and, while nothing completely new was added to the way Java operates in the low-level design, the high-level design and programming techniques were undoubtedly changed. The result was, for example, a rise in annotation-based frameworks and simpler programming as a whole.

Java 8 has been released with lambda expressions support, type annotations, and parameter reflection for public use. But it was possible to use it since late 2012, at least. It was possible to have all this functionality, to write programs with all these features, to have fun with testing new technologies, well before the official release date. Some enterprise developers consider modern Java as unstable and slightly unpredictable even after the release. However, each programmer, who was interested in the new technologies' testing and support and who contributes to it, was able to test and try OpenJDK 8 when it was at development stage.


What is the situation with OpenJDK 9 early access previews were released immediately after the Java 8 release. So we can now try OpenJDK 9. Of course, it is still unstable and doesn't even pass some regression tests, but it is going to change.

What are the main differences between OpenJDK 8 and 9?


There are three main features:

- ▶ The First is the long awaited elimination of type erasure. With new refined generics, it will be possible to determine which type is used in a collection or map, or a tuple—on every generic reference. For all Java programmers this is going to be a major relief.
- ▶ The second feature is intended to bring the Java platform a new ally—the whole might of a GPU will now be in the hands of a programmer, using only standard features without a scrap of native code. It will be explained further in this chapter.
- ▶ And a third one is Graal, the project that exposes Java VM APIs to the end user. It is a great breakthrough since it is possible to change the way Java operates on the fly.

There is more to be done; Java 9 will also contain less GC types without a drop in performance.

[ Be aware, underscore (_) will not be a legal identifier name in Java 9, so prepare your code in time. Find more details at <http://openjdk.java.net/jeps/213>.]

Also, for those who work with money transactions and financial analytics, there is another Java 9 feature—JSR 354, the Money API. It will implement the ISO-4217 standard currencies along with some additional ones. Currency arithmetics will also be introduced.

[ To test the Java Money API, build the source code from <https://github.com/JavaMoney/jsr354-api>. However, this is a Maven project that meets the JSR requirements, and is not a part of the OpenJDK project.]

Building OpenJDK 9 on Mac OS X using Clang

At the time of writing, the OpenJDK 9 project was still quite similar to OpenJDK 8. So most of the information about building OpenJDK 9 can be found in *Chapter 4, Building OpenJDK 8*.

One point that differentiates OpenJDK 9 from OpenJDK 8 is the usage of the Clang compiler on Mac OS X. Starting with Xcode Version 5, Clang became the official compiler on Mac OS X instead of GCC. There were plans to use it as an official compiler for OpenJDK 8 on Mac OS X but that switch was postponed to OpenJDK 9.

In this recipe, we will build the current codebase of OpenJDK 9 using Xcode 5 and the Clang compiler.

Getting ready

For this recipe, we will need a clean Mac OS X 10.8 Mountain Lion or 10.9 Mavericks running with Mercurial source control tools installed.

How to do it...

The following procedure will help us to build OpenJDK 9:

1. Download Xcode 5 from <https://developer.apple.com/xcode/> (an Apple developer's account is required, registration is free) and install it.
2. Download the Command Line Tools for the corresponding minor version of Xcode using the same download link mentioned previously, and install it.
3. Run the following command from the terminal to set up the Command Line Tools:

```
sudo xcode-select -switch /Applications/Xcode.app/Contents/Developer/
```
4. Install JDK 8—Oracle distribution, or prebuilt OpenJDK binaries may be used.
5. Obtain the source code from the Mercurial repository:

```
hg checkout http://hg.openjdk.java.net/jdk9/jdk9/
bash ./get_source.sh
```
6. Run the autotools configuration script:

```
bash ./configure --with-boot-jdk=path/to/jdk8
```
7. Start the build:

```
make all 2>&1
```

8. The built binaries will be put into the following directory:

```
build/macosx-x86_64-normal-server-release/images/j2sdk-image
```

How it works...

Xcode 5 uses the Clang compiler by default and OpenJDK 9 already has all the adjustments required for switching compiler from GCC to Clang.

There's more...

With the older versions of OpenJDK 9, the installation of the X11 server may be required. The X11 server can be installer from the *XQuartz* project.

See also

- Building OpenJDK 8 on Mac OS X recipe from *Chapter 4, Building OpenJDK 8*

Building OpenJDK 9 on Windows using MSYS

The Windows operating system has a long history of different tools providing a Unix-like environment. Tools such as Microsoft POSIX subsystem, Interix, Windows Services for UNIX, MKS Toolkit, Cygwin, MinGW/MSYS, and so on, existed during the various periods of Windows history and provided different levels of Unix compliance.

The three latter tools are most relevant to OpenJDK builds. MKS Toolkit was used for internal builds in Sun Microsystems because it provided better speed than Cygwin. Support for MKS Toolkit was discontinued with OpenJDK 7. Cygwin, that we described in detail in *Chapter 2, Building OpenJDK 6*, was used as the main and only tool to build OpenJDK 7 and 8.

MSYS (short form for minimal system) is a part of the MinGW (Minimalist GNU for Windows) project. The MinGW project was started as a fork of Cygwin with the goal to provide closer integration with Windows API for the cost of lower level Unix support. MinGW-based applications are standalone native Windows applications and do not require the `cygwin.dll` library. Among other things, this can bring better speed for some applications using Windows API through MinGW instead of emulated Unix calls (like `fork`) through Cygwin. Applications should be changed accordingly, though.

The MSYS project provides a minimalistic shell environment and also provides first-class support for running GNU Autoconf based builds. Actually running Autoconf's configure scripts efficiently was one of the goals of MSYS. In some cases, the configure scripts can be extremely slow in Cygwin because of extensive use of new processes spawning.

Due to better speed in OpenJDK builds, MSYS support was restored in OpenJDK 9 as a second supported environment along with Cygwin.

In this recipe, we will build OpenJDK 9 on Windows 7 using MSYS instead of Cygwin. At the time of writing, OpenJDK 9 was still in development and we have used the latest available source code.

Getting ready

For this recipe, we will need Windows 7 (32- or 64-bit) running.

How to do it...

The following procedure will help us to build OpenJDK 9:

1. Download Microsoft .NET Framework 4 from the Microsoft website and install it.
2. Download Microsoft Windows SDK for Windows 7 (the `GRMSDKX_EN_DVD.iso` file) from the Microsoft website and install it to the default location. The .NET Development and Common Utilities components are not required.
3. Download Visual Studio 2010 Express and install it (C++ variant) to the default location.
4. Download and install Microsoft DirectX 9.0 SDK (Summer 2004) to the default installation path. Note that this distribution is not available on the Microsoft website anymore. It may be downloaded elsewhere online, the file details are as follows:
name: `dxsdk_sum2004.exe`
size: `239008008 bytes`
sha1sum: `73d875b97591f48707c38ec0dbc63982ff45c661`
5. Download the prebuilt FreeType libraries from the `openjdk-unofficial-builds` GitHub project (directory `7_64`) and put the binaries into the `c:\freetype\lib` directory and the header files into the `c:\freetype\include` directory.
6. Install the OpenJDK 8 binaries or Oracle Java 8 into `c:\jdk8`.
7. Download and install the Mercurial SCM tool from the `mercurial.selenic.com` website.
8. Clone the current development forest of OpenJDK 9 to the `C:\openjdk` directory:
hg clone `http://hg.openjdk.java.net/jdk9/jdk9/`
9. Download the `mingw-get` utility (`mingw-get-setup.exe`) from `http://mingw.org/` and install it to the `C:\MinGW` path.
10. Run the `cmd.exe` shell and navigate to the directory.

11. Run the following command to revert the installed version of MSYS from the latest one to 1.0.17:

```
mingw-get install --reinstall --recursive msys-core=1.0.17-1
```
12. Run the following commands to install all required MSYS packages:

```
mingw-get install msys-zip  
mingw-get install msys-unzip  
mingw-get install msys-diffutils  
mingw-get install msys-file  
mingw-get install msys-mktemp  
mingw-get install msys-tar  
mingw-get install msys-xargs  
mingw-get install msys-findutils  
mingw-get install msys-make
```
13. Run the MSYS shell using the `C:\MinGW\msys\1.0\msys.bat` file.
14. Navigate to the `c/openjdk` directory and download the source code for all OpenJDK subrepositories with the following command:

```
bash ./get_source.sh
```
15. Add a path for the JDK 8 binaries in the `PATH` variable:

```
export PATH=/c/jdk8/bin:$PATH
```
16. Change the filesystem permissions for all source files:

```
chmod -R 777 .
```
17. Run the configure script specifying path to the FreeType binaries:

```
./configure --with-freetype=/c/freetype
```
18. Start the build process writing the output to the screen and logfile simultaneously:

```
make | tee make.log
```
19. Wait for the build to finish.

How it works...

At the time of writing, OpenJDK 9 code uses the same toolchain as OpenJDK 8, so the environment setup is similar.

We use Version 1.0.17 of MSYS because the regression related to multicores support appeared in the 1.0.18 Version. This regression is not fixed at the time of writing but will most likely be fixed in the following versions.

The `mingw-get` utility is a package manager that allows us to install or update the required MSYS packages.

See also

- ▶ The *Installing Cygwin for Windows builds* recipe from Chapter 2, *Building OpenJDK 6*
- ▶ The *Building OpenJDK 8 on Windows 7 SP1* recipe from Chapter 4, *Building OpenJDK 8*
- ▶ The OpenJDK bug related to the restoration of MSYS support in OpenJDK 9 at <https://bugs.openjdk.java.net/browse/JDK-8022177>
- ▶ The OpenJDK mailing list thread about the regression in MSYS 1.0.18 at <http://mail.openjdk.java.net/pipermail/build-dev/2014-August/012917.html>
- ▶ The MSYS website at <http://www.mingw.org/wiki/msys>
- ▶ Information about Windows service for Unix at <http://technet.microsoft.com/en-us/library/bb496506.aspx>
- ▶ The MKS Toolkit website at <http://mkssoftware.com/>
- ▶ The Cygwin website at <http://cygwin.com/>

Running and testing the early access preview of OpenJDK 9

We will get the newest OpenJDK code available and will test for feature availability. Don't hesitate to try new features, they may be available in the newer releases. Since the OpenJDK 9 release, it will remain the fastest way to give OpenJDK 9 a test. Hopefully, upon the release, the same thing will work for OpenJDK 10.

Getting ready

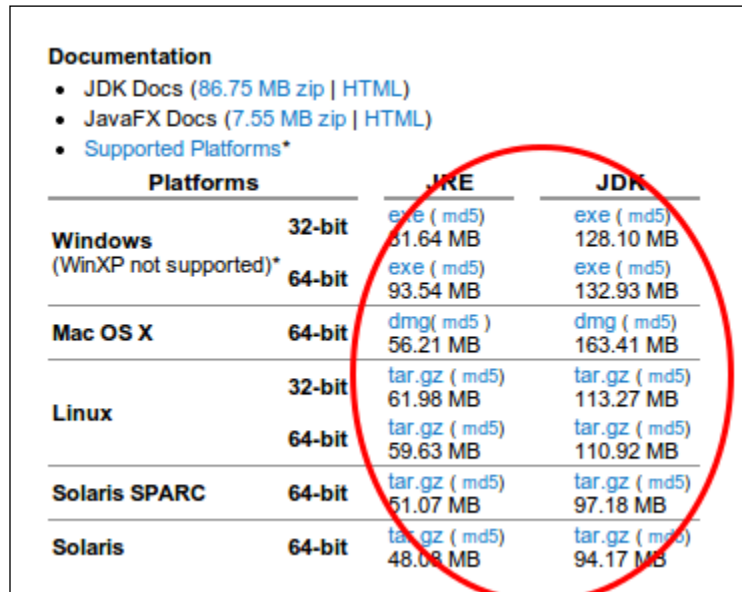
You will need an Internet connection. Aside from that, nothing is needed.

How to do it...

We will download, unpack, and run the latest publicly available full OpenJDK build:

1. Open the page <https://jdk9.java.net/>.

2. Download the early access preview, as shown:



Documentation			
		JRE	JDK
Windows (WinXP not supported)*	32-bit	exe (md5) 31.64 MB	exe (md5) 128.10 MB
	64-bit	exe (md5) 93.54 MB	exe (md5) 132.93 MB
Mac OS X	64-bit	dmg (md5) 56.21 MB	dmg (md5) 163.41 MB
Linux	32-bit	tar.gz (md5) 61.98 MB	tar.gz (md5) 113.27 MB
	64-bit	tar.gz (md5) 59.63 MB	tar.gz (md5) 110.92 MB
Solaris SPARC	64-bit	tar.gz (md5) 51.07 MB	tar.gz (md5) 97.18 MB
Solaris	64-bit	tar.gz (md5) 48.08 MB	tar.gz (md5) 94.17 MB

3. Run an installer.
4. You can find out how to install OpenJDK from an archive in *Chapter 1, Getting Started with OpenJDK* of this book.

Test some interesting features that are already included in the early access JDK. Look at the following code:

```
DatagramSocket socket = new DatagramSocket(4445);  
System.out.println(socket.supportedOptions());
```

When executed, you may expect it to return the following string, or something similar:

```
[SO_SNDBUF, SO_RCVBUF, SO_REUSEADDR, IP_TOS]
```

Although there are some minor improvements and a lot of bug fixes, there are not yet any major changes in the early access preview.

How it works...

In the source repository on <http://hg.openjdk.java.net/jdk9/jdk9>, there are tags, such as `jdk9-b<build number>`, which are automatically built into the early access releases. Although there are no nightly builds, you can always build them from source, if you have lots of time and a machine that's powerful enough to start with.

Don't forget to update your once installed releases—there will be really major and exciting changes, including those explained next. Sooner or later, developers will come out with Java 9 full, and then there will be time to test it before it becomes production ready.

There's more...

You can also build OpenJDK 9 from source:

1. Clone the source code repository hg clone at `http://hg.openjdk.java.net/jdk9/jdk9`
2. Get the source code of the OpenJDK subprojects:

```
chmod 755 ./get_source.sh && ./get_source.sh
cd jigsaw
```
3. Then configure the OpenJDK instance to be built:

```
chmod 755 ./configure.sh && ./configure.sh
```
4. And, finally, do the build itself:

```
Make
```
5. The final strings of your output will look like this:

```
-----
Finished building OpenJDK for target 'default'
```

Using Jigsaw

Jigsaw is the brand-new modular system for Java. It brings to mind some existing products, such as Maven or Gradle, but its most interesting feature is the possibility of the modularization of the JDK itself. Jigsaw will allow, upon its full completion, to modularize even some features that was thought as unseparable, such as HotSpot binaries.

Jigsaw incorporates proposals about the Java modular system. Modularity means scalability—from small, embedded devices that need only basic functionality and have poor performance, to full-scale data centers with dozens of machines. Some of these goals have already been reached—but Jigsaw presents a universal way to resolve dependencies on all platforms, starting from Java itself.

Several JEPs are part of Jigsaw:

- ▶ **JEP 200:** This makes the JDK itself modular
- ▶ **JEP 201:** This makes the source code modular
- ▶ **JEP 220:** This makes the runtime Java images modular, so they can be loaded in parts

Some information about the progress of the JEPs are found at the following JIRA links:

- ▶ <https://bugs.openjdk.java.net/browse/JDK-8051619>
- ▶ <https://bugs.openjdk.java.net/browse/JDK-8051618>
- ▶ <https://bugs.openjdk.java.net/browse/JDK-8061971>

Also, the core of Jigsaw is JSR 376—the Java platform Module System.

Getting ready

You will need Internet access. Also, some experience with Maven, or similar software is desired. Any knowledge about how the build systems work from the inside will be appreciated.

How to do it...

The following procedure will teach you how to build Jigsaw-enabled Java:

1. First, let's clone a source repo hg clone from <http://hg.openjdk.java.net/jigsaw/jigsaw>.
2. Then, let's get the source code of the OpenJDK subprojects:

```
chmod 755 ./get_source.sh && ./get_source.sh  
cd jigsaw
```
3. Then configure the OpenJDK instance to be built:

```
chmod 755 ./configure && ./configure
```
4. And, finally, do the build itself:

```
make all
```

```
Finished building OpenJDK for target 'default'
```

Congratulations, you've built Jigsaw-enabled Java.

Now, we will do some tests:

1. Let's consider the simple *helloworld1* program:

```
package me.dsmd.jigsawsample
import me.dsmd.helloworldstrings.Strings
public class Main{
    public void main(String [] args){
        System.out.println(Strings.helloworld1());
    }
}
```

2. It has one class, which is imported from a yet nonexistent package.
3. Let's create it.

```
package me.dsmd.helloworldstrings
public class Strings{
    public String helloworld1(){
        return "Hello World";
    }
}
```

Now, we will try to link it using Jigsaw.

Jigsaw stores the module declaration in the file named `module-info.java`.

1. Let's create it for those two packages as follows:

```
module me.dsmd.jigsawsample @ 1.0{
    requires me.dsmd.helloworldstrings
    class me.dsmd.jigsawsample.Main
}
module me.dsmd.helloworldstrings @ 0.1 {
    class me.dsmd.helloworldstrings.Strings
}
```

These files are to be placed in the root directory of a package.

2. Let's consider a situation when all those modules are placed in the same directory named `src`:

```
.
├── modules
└── src
    └── me.dsmd.helloworldstrings
```

```
|   |— me
|   |   └─ dsmd
|   |       └─ helloworldstrings
|   |           └─ Strings.java
|   └─ module-info.java
└─ me.dsmd.jigsawsample
    |   |— me
    |   |   └─ dsmd
    |   |       └─ jigsawsample
    |   |           └─ Main.java
    └─ module-info.java
```

3. Then, let's compile them with `javac` from your jigsaw build:

```
javac -verbose -d modules -modulepath modules -sourcepath \ `find
src -name '*.java'`
```

How it works...

Jigsaw is a modular system that gives Java its very own build system. Similar systems were spawned in the Java world long ago, but they lacked the core support. They were never able to bring the modular support advantages to Java's own features. Of course, there are some downsides as well. The *Write once, run everywhere* slogan is not so applicable as it was before.

We use the newly built OpenJDK commands to create, install, and export modules. Those commands are still under heavy development, but the specification is already written so, hopefully, nothing will significantly change before the production-access release.

There's more...

You also can install a module as a library. To do so, run the following command:

- To create a module library:

```
$ jmod -L lib1 create
```

This creates a module library `lib1`.

- To install some modules to the library:

```
$ jmod -L lib1 install modules module1 module2
```

This will install your modules under the system parent in the library `lib1`.

Currently, there is no way to remove the module from the library. Maybe, there will not be any in the release either. For now, the simplest way to remove a module from the library is to delete it physically from the repository:

```
rm -rf lib1/module1
```

Also, you can run a module, if it contains a standard entry point:

```
java -L lib1 -m module1
```



The new `-m` option is also contained only in the Jigsaw-enabled Java command. As of now (June 2014) it is not contained in any public early access preview.

As a next feature, you can export a module as a file by performing the following command:

```
jpkg -m modules/module1 jmod module1
```

The `module1@<module version>.jmod` file will be created. It will contain the exported, ready-to-use module.

Building OpenJDK 9 with Graal

As described in the project page, Graal is:

A quest for the JVM to leverage its own J.

Upon completion of this project, JVM functions will be exposed via Java APIs, so the end user will be able to have access to the most low-level manipulation. It will be possible to write a Java compiler in Java, for example. Now, we will try to give it a test.

There is also **Truffle**, a framework that allows you to build your own language using Graal VM. It builds upon a notion of an **abstract syntax tree (AST)**, and the process, in fact, is really simple. To have a better look, see the following link:

<https://cesquivias.github.io/blog/2014/10/13/writing-a-language-in-truffle-part-1-a-simple-slow-interpreter/>

Getting ready

You will need an Internet connection. Also, it's recommended to read the chapters about building OpenJDK.

How to do it...

Have a look at the following procedure to build OpenJDK with Graal:

1. First, clone a source repository:

```
hg clone http://hg.openjdk.java.net/graal/graal
cd graal
export EXTRA_JAVA_HOMES=/usr/lib/jvm/java-1.7.0-openjdk
./mx.sh build
```

2. Enter a selection value, upon which `vm graal` will be built. Unfortunately, it will not be built without modification against the OpenJDK 9-ea preview:

```
./mx.sh build
[1] /usr/lib/jvm/java-1.7.0-openjdk
[2] /usr/lib/jvm/java-1.6.0-openjdk-amd64
[3] /usr/lib/jvm/java-6-oracle
[4] /usr/lib/jvm/jdk1.9.0
[5] /usr/lib/jvm/java-7-oracle
[6] /usr/lib/jvm/java-8-oracle
[7] /usr/lib/jvm/java-6-openjdk-amd64
[8] /usr/lib/jvm/java-7-openjdk-amd64
[9] /usr/lib/jvm/java-1.7.0-openjdk-amd64
[10] <other>
```

3. Then choose which VM will be executed. There are two types of VM. In a nutshell, `server vm` will use the default hotspot compilation, using Graal itself only for explicit Graal API calls, while `graal vm` will compile everything through Graal. The first option is much more suitable for production VMs, while the second is favorable for testing purposes.

Please select the VM to be executed from the following:

```
[1] server - Normal compilation is performed with a tiered system
(C1
+ C2), Truffle compilation is performed with Graal. Use this
for
optimal Truffle performance.
[2] graal - Normal compilation is performed with a tiered system
(C1 +
Graal), Truffle compilation is performed with Graal.
```

4. Then, make a cup of tea, the process may take several dozen minutes.
5. Then, if you want to initialize your IDE project, run `./mx.sh ideinit`.
6. Then, open your favorite IDE and open a resulting project. It will be shown here with IntelliJ Idea:



7. Explore various tests.

How it works...

The Graal enabled VM will expose Java APIs to the end user.

There's more...

In *Chapter 8, Hacking OpenJDK*, we added new intrinsics to the HotSpot, using the `crc32` calculation as an example. In the Graal project, there is a similar test, which tests the compiled substitution of the `CRC32#updateByteBuffer` method. It is contained in the `com.oracle.graal.hotspot.jdk8.test` package. Run it, and enjoy the performance change.

Building OpenJDK 9 with Sumatra

For a long time, Java was considered as a primarily backend tool, due to its cross-platform vectoring. Only J2ME was capable of achieving long-term superiority in the mobile segment. But now it's going to change. Project Sumatra has the aim of delivering GPU-calculation standards to the Java guys.

Getting ready

You will probably need a GPU that supports CUDA and OpenGL, or a HSAIL simulator running (because on-board GPUs do not support native GPU languages).

How to do it...

Sumatra developers are making wide use of the Graal project, described earlier. The build consists of two stages. Firstly, the Sumatra JDK is built, like a normal OpenJDK build, as shown:

```
make
```

```
Building OpenJDK for target 'default' in configuration 'linux-x86_64-  
normal-server-release'
```

```
## Starting langtools
```

```
## Finished langtools (build time 00:00:03)
```

```
## Starting hotspot
```

```
## Finished hotspot (build time 00:00:01)
```

```
## Starting corba
```

```
## Finished corba (build time 00:00:00)
```

```
## Starting jaxp
```

```
## Finished jaxp (build time 00:00:01)
```

```
## Starting jaxws
```

```
## Finished jaxws (build time 00:00:02)
```

```
## Starting jdk
```

```
## Finished jdk (build time 00:00:21)
```

```
----- Build times -----
```

```
Start 2014-07-06 00:17:24
```

```
End   2014-07-06 00:17:52
```

```
00:00:00 corba
```

```
00:00:01 hotspot
```

```
00:00:01 jaxp
00:00:02 jaxws
00:00:21 jdk
00:00:03 langtools
00:00:28 TOTAL
```

```
-----
```

Finished building OpenJDK for target 'default'

The second stage consists of building a Graal JDK on top of the Sumatra JDK. It may be tricky, but hopefully it will work:

1. Clone a repository `hg clone http://hg.openjdk.java.net/sumatra/sumatra-dev/:`
`chmod 755 configure && ./configure`
2. Get the source code:
`chmod 755 get_source.sh && ./get_source.sh`
3. Then make the source code:
`make`
4. Export `JAVA_HOME` to the newly built OpenJDK instance:
`export JAVA_HOME=<path-to-sumatra-dev>/build/linux-<your-arch>-normal-server-release/images/j2sdk-image`
5. Build the HSAIL-enabled graal:
`/mx.sh --vmbuild product --vm server build`

Congratulations! You have a Sumatra-enabled VM.

Let's do a little test. Consider a code from an official sample:

```
package simple;

import java.util.stream.IntStream;

public class Simple {

    public static void main(String[] args) {
        final int length = 8;
        int[] ina = new int[length];
        int[] inb = new int[length];
```



```
int[] out = new int[length];

// Initialize the input arrays - this is offloadable
IntStream.range(0, length).parallel().forEach(p -> {
    ina[p] = 1;
    inb[p] = 2;
});

// Sum each pair of elements into out[] - this is offloadable
IntStream.range(0, length).parallel().forEach(p -> {
    out[p] = ina[p] + inb[p];
});

// Print results - this is not offloadable since it is
// calling native code etc.
IntStream.range(0, length).forEach(p -> {
    System.out.println(out[p] + ", " + ina[p] + ", " +
inb[p]);
});
}
```

It contains two off-loadable lambdas. We will try to make them run in parallel, using the HSA API.

1. First, set `JAVA_HOME` to the Graal JDK:
`export JAVA_HOME=/path/to/graal/jdk1.8.0-internal/product/`
2. Then clone the OKRA HSA interface:
`git clone https://github.com/HSAFoundation/Okra-Interface-to-HSA-Device.git`
3. Make it runnable:
`export PATH=$PATH:/path/to/okra/dist/bin`
`export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path/to/okra/dist/bin`
4. Run the example with and without offloading.
5. You will get the following code in your terminal:

```
$JAVA_HOME/bin/java -server -esa -XX:+TraceGPUInteraction -Dcom.
amd.sumatra.offload.immediate=true -G:Log=CodeGen simple.Simple
...
[HSAIL] library is libokra_x86_64.so
```

```

[GPU] registered initialization of Okra (total initialized: 2)
[CUDA] Ptx::get_execute_kernel_from_vm_address
[thread:1] scope:
  [thread:1] scope: GraalCompiler
    [thread:1] scope: GraalCompiler.CodeGen
    Nothing to do here
    Nothing to do here
    Nothing to do here
    version 0:95: $full : $large;
// static method HotSpotMethod<Simple.lambda$main$0(int[], int[],
int)>
kernel &run (
  align 8 kernarg_u64 %_arg0,
  align 8 kernarg_u64 %_arg1
) {
  ld_kernarg_u64 $d0, [%_arg0];
  ld_kernarg_u64 $d1, [%_arg1];
  workitemabsid_u32 $s0, 0;

@L0:
  cmp_eq_b1_u64 $c0, $d1, 0; // null test
  cbr $c0, @L1;

@L2:
  ld_global_s32 $s1, [$d1 + 12];
  cmp_ge_b1_u32 $c0, $s0, $s1;
  cbr $c0, @L8;

@L3:
  cmp_eq_b1_u64 $c0, $d0, 0; // null test
  cbr $c0, @L4;

@L5:
  ld_global_s32 $s1, [$d0 + 12];
  cmp_ge_b1_u32 $c0, $s0, $s1;
  cbr $c0, @L7;

@L6:
  cvt_s64_s32 $d2, $s0;
  mul_s64 $d2, $d2, 4;

```

```
        add_u64 $d0, $d0, $d2;
        st_global_s32 1, [$d0 + 16];
        cvt_s64_s32 $d0, $s0;
        mul_s64 $d0, $d0, 4;
        add_u64 $d1, $d1, $d0;
        st_global_s32 2, [$d1 + 16];
        ret;

@L1:
        mov_b32 $s0, -6155;

@L9:
        ret;

@L4:
        mov_b32 $s0, -4363;
        brn @L9;

@L8:
        mov_b32 $s0, -6683;
        brn @L9;

@L7:
        mov_b32 $s0, -4891;
        brn @L9;
};

[HSAIL] heap=0x00007f4c9801de38
[HSAIL] base=0x05a00000, capacity=209190912
External method:simple.Simple.lambda$main$0([I[II)V
installCode0: ExternalCompilationResult
[HSAIL] sig:([I[II)V args length=2, _parameter_count=3
[HSAIL] static method
[HSAIL] HSAILKernelArguments::do_array, _index=0, 0x82b20888, is a
[I
[HSAIL] HSAILKernelArguments::do_array, _index=1, 0x82b208b8, is a
[I
[HSAIL] HSAILKernelArguments::not pushing trailing int
[thread:1] scope: GraalCompiler
[thread:1] scope: GraalCompiler.CodeGen
Nothing to do here
```

```

        Nothing to do here
        Nothing to do here
        version 0:95: $full : $large;
// static method HotSpotMethod<Simple.lambda$main$1(int[], int[],
int[], int)>
kernel &run (
    align 8 kernarg_u64 %_arg0,
    align 8 kernarg_u64 %_arg1,
    align 8 kernarg_u64 %_arg2
) {
    ld_kernarg_u64 $d0, [%_arg0];
    ld_kernarg_u64 $d1, [%_arg1];
    ld_kernarg_u64 $d2, [%_arg2];
    workitemabsid_u32 $s0, 0;

@L0:
    cmp_eq_b1_u64 $c0, $d0, 0; // null test
    cbr $c0, @L1;

@L2:
    ld_global_s32 $s1, [$d0 + 12];
    cmp_ge_b1_u32 $c0, $s0, $s1;
    cbr $c0, @L12;

@L3:
    cmp_eq_b1_u64 $c0, $d2, 0; // null test
    cbr $c0, @L4;

@L5:
    ld_global_s32 $s1, [$d2 + 12];
    cmp_ge_b1_u32 $c0, $s0, $s1;
    cbr $c0, @L11;

@L6:
    cmp_eq_b1_u64 $c0, $d1, 0; // null test
    cbr $c0, @L7;

@L8:
    ld_global_s32 $s1, [$d1 + 12];
    cmp_ge_b1_u32 $c0, $s0, $s1;
    cbr $c0, @L10;

```

```
@L9:
    cvt_s64_s32 $d3, $s0;
    mul_s64 $d3, $d3, 4;
    add_u64 $d1, $d1, $d3;
    ld_global_s32 $s1, [$d1 + 16];
    cvt_s64_s32 $d1, $s0;
    mul_s64 $d1, $d1, 4;
    add_u64 $d2, $d2, $d1;
    ld_global_s32 $s2, [$d2 + 16];
    add_s32 $s2, $s2, $s1;
    cvt_s64_s32 $d1, $s0;
    mul_s64 $d1, $d1, 4;
    add_u64 $d0, $d0, $d1;
    st_global_s32 $s2, [$d0 + 16];
    ret;

@L1:
    mov_b32 $s0, -7691;

@L13:
    ret;

@L4:
    mov_b32 $s0, -6411;
    brn @L13;

@L10:
    mov_b32 $s0, -5403;
    brn @L13;

@L7:
    mov_b32 $s0, -4875;
    brn @L13;

@L12:
    mov_b32 $s0, -8219;
    brn @L13;

@L11:
    mov_b32 $s0, -6939;
    brn @L13;

};
```

```

[HSAIL] heap=0x00007f4c9801de38
[HSAIL] base=0x05a00000, capacity=209190912
External method:simple.Simple.lambda$main$1([I[I[II)V
installCode0: ExternalCompilationResult
[HSAIL] sig:([I[I[II)V  args length=3, _parameter_count=4
[HSAIL] static method
[HSAIL] HSAILKernelArguments::do_array, _index=0, 0x82b208f8, is a
[I
[HSAIL] HSAILKernelArguments::do_array, _index=1, 0x82b20888, is a
[I
[HSAIL] HSAILKernelArguments::do_array, _index=2, 0x82b208b8, is a
[I
[HSAIL] HSAILKernelArguments::not pushing trailing int
3, 1, 2
3, 1, 2
3, 1, 2
3, 1, 2
3, 1, 2
3, 1, 2
3, 1, 2
3, 1, 2
3, 1, 2
3, 1, 2

```

How it works...

Sumatra runs on top of the Graal project. Since all operations with GPU are implemented on the VM level, Sumatra uses Graal to gain access to them. Sumatra features are at the heavy development stage, and they are subject to various unpredictable changes.

But the end user can use some of them, even now, to gain a new level of Java productivity at the expense of some compatibility and standardization.

There's more...

In the Graal suite, there is a possibility to test the Sumatra HSAIL feature.

To do so, run the following code:

```

./mx.sh --vm server unittest -XX:+TraceGPUInteraction -XX:+GPUOffload
-G:Log=CodeGen hsail.test.IntAddTest

```

The output should look like the following (for Linux Mint 15, or for other distributions\OS, results may slightly differ):

```
[HSAIL] library is libokra_x86_64.so
[HSAIL] using _OKRA_SIM_LIB_PATH_=/tmp/okraresource.
dir_2488167353114811077/libokra_x86_64.so
[GPU] registered initialization of Okra (total initialized: 2)
[CUDA] Ptx::get_execute_kernel_from_vm_address
JUnit version 4.8
.[thread:1] scope:
  [thread:1] scope: GraalCompiler
    [thread:1] scope: GraalCompiler.CodeGen
      Nothing to do here
      Nothing to do here
      Nothing to do here
      version 0:95: $full : $large;
// static method HotSpotMethod<IntAddTest.run(int[], int[], int[], int)>
kernel &run (
  align 8 kernarg_u64 %_arg0,
  align 8 kernarg_u64 %_arg1,
  align 8 kernarg_u64 %_arg2
) {
  ld_kernarg_u64 $d0, [%_arg0];
  ld_kernarg_u64 $d1, [%_arg1];
  ld_kernarg_u64 $d2, [%_arg2];
  workitemabsid_u32 $s0, 0;
@L0:
  cmp_eq_b1_u64 $c0, $d0, 0; // null test
  cbr $c0, @L1;
@L2:
  ld_global_s32 $s1, [$d0 + 12];
  cmp_ge_b1_u32 $c0, $s0, $s1;
  cbr $c0, @L12;
@L3:
  cmp_eq_b1_u64 $c0, $d2, 0; // null test
  cbr $c0, @L4;
@L5:
```

```
ld_global_s32 $s1, [$d2 + 12];
cmp_ge_b1_u32 $c0, $s0, $s1;
cbr $c0, @L11;

@L6:
cmp_eq_b1_u64 $c0, $d1, 0; // null test
cbr $c0, @L7;

@L8:
ld_global_s32 $s1, [$d1 + 12];
cmp_ge_b1_u32 $c0, $s0, $s1;
cbr $c0, @L10;

@L9:
cvt_s64_s32 $d3, $s0;
mul_s64 $d3, $d3, 4;
add_u64 $d1, $d1, $d3;
ld_global_s32 $s1, [$d1 + 16];
cvt_s64_s32 $d1, $s0;
mul_s64 $d1, $d1, 4;
add_u64 $d2, $d2, $d1;
ld_global_s32 $s2, [$d2 + 16];
add_s32 $s2, $s2, $s1;
cvt_s64_s32 $d1, $s0;
mul_s64 $d1, $d1, 4;
add_u64 $d0, $d0, $d1;
st_global_s32 $s2, [$d0 + 16];
ret;

@L1:
mov_b32 $s0, -7691;

@L13:
ret;

@L4:
mov_b32 $s0, -6411;
brn @L13;

@L10:
mov_b32 $s0, -5403;
brn @L13;

@L7:
```



```
    mov_b32 $s0, -4875;
    brn @L13;
@L12:
    mov_b32 $s0, -8219;
    brn @L13;
@L11:
    mov_b32 $s0, -6939;
    brn @L13;
};
```

```
[HSAIL] heap=0x00007f95b8019cc0
[HSAIL] base=0x05a00000, capacity=210763776
External method:com.oracle.graal.compiler.hsail.test.IntAddTest.
run([I[I[II)V
installCode0: ExternalCompilationResult
[HSAIL] sig:([I[I[II)V args length=3, _parameter_count=4
[HSAIL] static method
[HSAIL] HSAILKernelArguments::do_array, _index=0, 0x82b21970, is a [I
[HSAIL] HSAILKernelArguments::do_array, _index=1, 0x82b477f0, is a [I
[HSAIL] HSAILKernelArguments::do_array, _index=2, 0x82b479e0, is a [I
[HSAIL] HSAILKernelArguments::not pushing trailing int
```

Time: 0.153

OK (1 test)

The completion of this test will mean that the HSAIL functions are working OK, so the cutting-edge Java already benefits from your GPU.

13

Build Automation

In this chapter, we will cover:

- ▶ Installing VirtualBox
- ▶ Preparing SSH keys
- ▶ Preparing VirtualBox machines with Linux
- ▶ Preparing VirtualBox machines with Mac OS X
- ▶ Preparing VirtualBox machines with Windows
- ▶ Automating builds
- ▶ Building cross-platform installers

Introduction

Automated builds are widely used in cross-platform software development. *Build farm* with build machines for each supported operating systems is required to build projects remotely and run tests on all platforms. With the rising popularity of software virtualization tools, it became possible to deploy a *virtual build farm* using a single physical machine.

Besides the build environment in each OS, the essential part of an automated build is the communication between the *master* machine and the *build* machines. The following communication jobs might be required:

- ▶ The master should prepare and start the build machine
- ▶ The master should send the source code directly to the build machine or fire sources fetching process from the build machine itself
- ▶ The master should start the build process
- ▶ The build machine should send build logs to the master during the build

- ▶ The build machine should send result binaries to the master
- ▶ The master should shut down the build machine

In this chapter, we will prepare the OpenJDK build environment for Windows, Linux, and Mac OS X. This task can be done using a high-level build automation (or Continuous Integration) tool. However, such tools can be limited in functionality and inflexible for our task. While all such tools should do similar jobs (as listed previously), different tools can have different configurations and peculiarities, and the knowledge of one tool can be less useful for another. Also, such tools bring in an additional level of complexity with possible tool-specific problems.

You will learn how to perform build automation using the most basic tools for a better understanding of the process. The bash shell is already used for OpenJDK builds on all platforms (natively on Linux/Mac and through Cygwin on Windows), so we will use bash scripts for setting up and starting to build virtual machines. For communication (sending commands and data), we will use the SSH protocol; implementation for this is usually preinstalled on Unix-like operating systems and can also be installed on Windows. For virtualization, we will use the popular VirtualBox tool from Oracle.

Installing VirtualBox

VirtualBox is a popular virtualization toolbox from Oracle Corporation that allows us to run *virtual* instances of other operating systems on top of the *host* operating system. In this recipe, we will install VirtualBox and configure host network interfaces. Such a configuration will allow us to connect from host to guest and back during the automated builds.

Getting ready

For this recipe, we will require the Ubuntu 12.04 amd64 operating system running.

How to do it...

The following procedure will help you to install VirtualBox:

1. Download the installation package from the VirtualBox website (<https://www.virtualbox.org/>) and install it.
2. Install the virtual network interface package:

```
sudo apt-get install uml-utilities
```
3. Create a virtual interface `tap0` that will be used for connections to and from the guest machine:

```
sudo tuncctl -t tap0 -u your_username
```

4. Configure the interface `tap0`:

```
sudo ifconfig tap0 192.168.42.2 up dstaddr 192.168.42.1 netmask 255.255.255.0
```
5. Create an arbitrary VirtualBox machine using GUI forms.
6. Navigate to **Settings | Network form** and check whether the **tap0** network interface is available in the interfaces drop-down list when the **Bridged Adapter** mode is used.

How it works...

VirtualBox supports different networking options for guest machines. One of them—Bridged Adapter—allows us to connect from host to guest and back using static IP addresses. To set up this mode on the host side, we need an additional virtual network interface with a separate address. The `tunctl` utility that comes as part of the `uml-utilities` package allows us to create a virtual interface such as `tap0`.

There's more...

Multiple virtual interfaces can be created with different addresses to run multiple virtual machines simultaneously. Mac OS X can be used as a host machine, the `tuntaposx` kernel extensions are required to use the `tunctl` utility.

See also

- ▶ The Oracle VirtualBox user manual at <https://www.virtualbox.org/manual/UserManual.html>

Preparing SSH keys

In this chapter, we will use virtual machines to build OpenJDK. During the build source code, control commands, logs, and result binaries should be sent between host and virtual machines. We will use the ubiquitous **Secure Shell (SSH)** protocol and its most popular implementation **OpenSSH** for these tasks.

SSH allows us to send data between the machines and run the commands remotely. When the client performs an SSH connection, it should be authenticated against the server. Besides the user/password authentication, OpenSSH also supports authentication using asymmetric cryptography (RSA or similar) keys. With SSH keys configured, a client can connect to server without manual intervention. This eases the scripting for copying multiple files or running remote tasks.

In this recipe, we will prepare a set of public and private keys and use these on all virtual machines and on the host machine during the build.

Getting ready

For this recipe, we will need a Unix-like operating system running with an OpenSSH server and client installed. For example, the Ubuntu 12.04 operating system can be used.

How to do it...

The following procedure will help us to install VirtualBox:

1. Generate client RSA keys pairs for the host and guest machines:

```
ssh-keygen -q -P "" -t rsa -f vmhost__id_rsa
ssh-keygen -q -P "" -t rsa -f vmguest__id_rsa
```
2. Generate server RSA key pairs for the host and guest machines:

```
ssh-keygen -q -P "" -t rsa -f vmhost__ssh_host_rsa_key
ssh-keygen -q -P "" -t rsa -f vmguest__ssh_host_rsa_key
```
3. Create a new user on the host machine that will be used to manage builds, and login under this user:

```
sudo adduser packt
```
4. Run VirtualBox and configure networking as described in the previous recipe, *Installing VirtualBox*, with the 192.168.42.2 host IP address and the 192.168.42.1 guest IP address.
5. Create a user with the same name on the guest machine and login under this user:

```
sudo adduser packt
```
6. Check whether the `ping` command works successfully from host to guest and back from guest to host:

```
ping 192.168.42.1
PING 192.168.42.2 (192.168.42.2) 56(84) bytes of data.
64 bytes from 192.168.42.2: icmp_req=1 ttl=64 time=0.181 ms
...
```
7. On the host machine, set up the client keys:

```
cp vmhost__id_rsa.pub ~/.ssh/id_rsa.pub
cp vmhost__id_rsa ~/.ssh/id_rsa
chmod 600 ~/.ssh/id_rsa
```

8. On the guest machine, set up server keys:


```
sudo rm -rf /etc/ssh/ssh_host_*
sudo cp vmguest__ssh_host_rsa_key.pub /etc/ssh/ssh_host_rsa_key.pub
sudo cp vmguest__ssh_host_rsa_key /etc/ssh/ssh_host_rsa_key
sudo chmod 600 /etc/ssh/ssh_host_rsa_key
```
9. On the guest machine, set up the host client public key:


```
cp vmhost__id_rsa.pub ~/.ssh/authorized_keys2
```
10. On the host machine, try to connect to the guest and confirm the new guest server key:


```
ssh packt@192.168.42.1
```
11. On the host machine, save the obtained fingerprint of the guest server key:


```
cp ~/.ssh/known_hosts vmhost__known_hosts
```
12. On the host machine, check whether we can now connect from host to guest without any passwords or additional confirmations:


```
ssh packt@192.168.42.1
```
13. Repeat steps 7 to 11, swapping host and guest sides to set up connections from guest to host.
14. Save the following keys for use later during the builds:
 - ❑ vmhost__id_rsa: This is the host client private key
 - ❑ vmhost__id_rsa.pub: This is the host client public key
 - ❑ vmhost__ssh_host_rsa_key: This is the host server private key
 - ❑ vmhost__ssh_host_rsa_key.pub: This is the host server public key
 - ❑ vmhost__known_hosts: This is the guest server key fingerprint to be used on the host
 - ❑ vmguest__id_rsa: This is the guest client private key
 - ❑ vmguest__id_rsa.pub: This is the guest client private key
 - ❑ vmguest__ssh_host_rsa_key: This is the guest server private key
 - ❑ vmguest__ssh_host_rsa_key.pub: This is the guest server public key
 - ❑ vmguest__known_hosts: This is the host server key fingerprint to be used on guest

How it works...

The `ssh-keygen` command generates a pair of asymmetric cryptography (in our example, RSA) keys.

SSH supports passwordless authentication based on keys. We prepared the set of keys that can be loaded to host and guest side (for all guest machines) to allow seamless connections from host to guest and back. So, now we can call a script on the host machine that will connect (or send files) to the guest and will be able to connect back to the host from the same guest session.

All keys are deliberately generated with an empty passphrase to allow connections without manual passphrase input.

There's more...

Connections over SSH are secure and this can be useful if you want to perform builds using remote machines instead of local virtual ones. If security is not required, then other protocols can be used. They do not require authentication or a keys setup, for example, some custom protocol over HTTP with support for commands and sending files.

The DSA or ECDSA keys can be used instead of the RSA keys.

A shell automation tool like `expect` can be used to set up automated connections with passwords instead of client keys.

See also

- ▶ The *Installing VirtualBox* recipe
- ▶ The OpenSSH manual on key generation that is available at <https://www.virtualbox.org/manual/UserManual.html>

Preparing VirtualBox machines with Linux

Many Linux-based operating systems have decent support for virtualization using VirtualBox. They also usually have an OpenSSH client and server preinstalled or available in the main packages repository.

In this recipe, we will set up an Ubuntu Linux virtual machine that can be used for automated OpenJDK builds.

Getting ready

For this recipe, we will require the Ubuntu 12.04 amd64 operating system with VirtualBox installed and a virtual network interface configured.

How to do it...

The following procedure will help us to prepare the Linux virtual machine:

1. Prepare the SSH keys as described in the recipe *Preparing SSH keys*.
2. Download the Ubuntu 12.04 server amd64 image from the Ubuntu website (<http://www.ubuntu.com/>).
3. In VirtualBox, create a virtual machine instance using the IDE storage controller and default values for other settings.
4. Install Ubuntu on to the virtual machine, set up networking as described in the recipe, *Installing VirtualBox*, and boot the virtual machine.
5. Create a user with the same name on the host machine, and login under this user:

```
sudo adduser packt
```
6. Set up the client keys:

```
cp vmguest__id_rsa.pub ~/.ssh/id_rsa.pub
cp vmguest__id_rsa ~/.ssh/id_rsa
chmod 600 ~/.ssh/id_rsa
```
7. Set up the server keys:

```
sudo rm -rf /etc/ssh/ssh_host_*
sudo cp vmguest__ssh_host_rsa_key.pub /etc/ssh/ssh_host_rsa_key.pub
sudo cp vmguest__ssh_host_rsa_key /etc/ssh/ssh_host_rsa_key
sudo chmod 600 /etc/ssh/ssh_host_rsa_key
```
8. Set up the host client public key:

```
cp vmhost__id_rsa.pub ~/.ssh/authorized_keys2
```
9. Set up the host key fingerprint:

```
cp vmguest__known_hosts ~/.ssh/known_hosts
```
10. Check whether the connection from host to guest and back works seamlessly:

```
ssh packt@192.168.42.1
```
11. Complete the manual build of OpenJDK using the recipe, *Building OpenJDK 8 Ubuntu Linux 12.04 LTS*, from Chapter 4, *Building OpenJDK 8*.

How it works...

In this recipe, we created a virtual machine instance with Ubuntu Linux and configured SSH keys to enable seamless automated connections to it.

The manual build was done on this VM to ensure all environment the is correct.

There's more...

Other Linux-based OSs can be used instead of Ubuntu 12.04. Other protocols/tools can be used for interaction between the host and guest machine instead of OpenSSH.

See also

- ▶ The *Building OpenJDK 8 Ubuntu Linux 12.04 LTS* recipe from *Chapter 4, Building OpenJDK 8*
- ▶ The *Installing VirtualBox* recipe
- ▶ The *Preparing SSH keys* recipe
- ▶ The Oracle VirtualBox user manual at <https://www.virtualbox.org/manual/UserManual.html>

Preparing VirtualBox machines with Mac OS X

Modern versions of the Mac OS X operating system support run in a virtualized environment using VirtualBox. Instructions to prepare the Mac OS X image for virtualization might differ vastly depending on the Mac OS X version and the host operating system. Exact instructions lie outside the scope of this book and can be found on the Internet.

Please note that running a guest Mac OS X on a non-Mac host operating system can violate your end user license agreement with Apple Inc., it's better to consult your lawyer about this.

Mac OS X has an OpenSSH client and server preinstalled.

Getting ready

For this recipe, we will require a ready-to-use Mac OS X image (for VirtualBox-VDI) Version 10.7 or later.

How to do it...

The following procedure will help us to prepare the Mac OS X virtual machine:

1. Prepare SSH keys as described in the *Preparing SSH keys* recipe in this chapter.
2. In VirtualBox, create a virtual machine instance using at least 2048 RAM, PIIX3 Chipset, disabled UEFI, single CPU, and IDE storage controller.
3. Set up networking as described in the *Installing VirtualBox* recipe in this chapter and boot the virtual machine.
4. Create a user with the same name on the host machine and login under this user.
5. Set up the client keys:

```
cp vmguest__id_rsa.pub ~/.ssh/id_rsa.pub
cp vmguest__id_rsa ~/.ssh/id_rsa
chmod 600 ~/.ssh/id_rsa
```

6. Set up the server keys:

```
sudo rm -rf /etc/ssh_host_*
sudo cp vmguest__ssh_host_rsa_key.pub /etc/ssh_host_rsa_key.pub
sudo cp vmguest__ssh_host_rsa_key /etc/ssh_host_rsa_key
sudo chmod 600 /etc/ssh_host_rsa_key
```

7. Set up the host client public key:

```
cp vmhost__id_rsa.pub ~/.ssh/authorized_keys2
```

8. Set up the host key fingerprint:

```
cp vmguest__known_hosts ~/.ssh/known_hosts
```

9. Check whether the connection from host to guest and back works seamlessly:

```
ssh packt@192.168.42.1
```

10. Complete the manual build of OpenJDK using the *Building OpenJDK 8 on Mac OS X* recipe from *Chapter 4, Building OpenJDK 8*.

How it works...

In this recipe, we created a virtual machine instance with Mac OS X and configured SSH keys to enable seamless automated connections to it.

The manual build was done on this VM to ensure that the environment setup was correct.

There's more...

In some Mac OS X versions, the preinstalled OpenSSH might not support the ECDSA SSH keys. This won't prevent us from finishing this recipe as we have used the RSA SSH keys. However, if you want to use ECDSA keys you can update the OpenSSH installation relatively easily using the Homebrew packaging system and its system duplicates repository, [homebrew/dupes](#).

Other protocols/tools can be used for interaction between the host and the guest machine instead of OpenSSH.

See also

- ▶ The *Building OpenJDK 8 on Mac OS X* recipe from *Chapter 4, Building OpenJDK 8*
- ▶ The *Installing VirtualBox* recipe
- ▶ The *Preparing SSH keys* recipe
- ▶ The Oracle VirtualBox user manual at <https://www.virtualbox.org/manual/UserManual.html>

Preparing VirtualBox machines with Windows

Popular virtualization tools have very good support for virtualizing Windows. The VirtualBox setup for Windows can be easier than with Mac OS X. However, the SSH protocol is less popular on Windows than on Unix-like operating systems, and the SSH server's setup on Windows might be complex.

In this recipe, you will learn how to set up a Windows virtual machine for automated builds. A set of in-depth instructions about configuration of free SSH servers on Windows will constitute a significant part of the recipe.

Getting ready

For this recipe, we will require a Windows 7 virtual machine VirtualBox image.

How to do it...

The following procedure will help us to prepare the Windows virtual machine:

1. Prepare the SSH keys as described in the *Preparing SSH keys* recipe in this chapter.

2. Download the Copssh SSH server implementation Version 3.1.4. Unfortunately, it was removed from public downloads by the authors, but still can be found on the Internet with these file details:

```
Copssh_3.1.4_Installer.exe  
size: 5885261 bytes  
sha1: faedb8ebf88285d7fe3e141bf5253cfa70f94819
```

3. Install Copssh into any Windows instance using the default installation parameters and copy the installed files somewhere for later usage.
4. In VirtualBox, create a virtual machine instance using the IDE storage controller and the default values for other settings.
5. Set up networking as described in the *Installing VirtualBox* recipe in this chapter and boot the virtual machine.
6. Create a user with the same name on the host machine and login under this user (we will use the name `packt`).
7. Download Windows Server 2003 Resource Kit Tools from the Microsoft website and extract `ntrights`, `instsrv`, and `srvany` utilities from it.
8. Copy the extracted Copssh files into the `c:\ssh` directory.
9. Set up users and rights for the SSH service using the following script:

```
net user sshd sshd /ADD  
net user SvcCOPSSH SvcCOPSSH /ADD  
net localgroup Administrators SvcCOPSSH /add  
ntrights +r SeTcbPrivilege -u SvcCOPSSH  
ntrights +r SeIncreaseQuotaPrivilege -u SvcCOPSSH  
ntrights +r SeCreateTokenPrivilege -u SvcCOPSSH  
ntrights +r SeServiceLogonRight -u SvcCOPSSH  
ntrights +r SeAssignPrimaryTokenPrivilege -u SvcCOPSSH  
ntrights +r SeDenyInteractiveLogonRight -u SvcCOPSSH  
ntrights +r SeDenyNetworkLogonRight -u SvcCOPSSH
```

10. Generate internal Copssh login and password information, and register Copssh as a Windows service:

```
c:\copssh\bin\mkpasswd -l > c:\obf\copssh\etc\passwd  
c:\copssh\bin\cygrunsrv.exe --install OpenSSHServer --args "-D"  
--path /bin/sshd --env "CYGWIN=binmode ntsec tty" -u SvcCOPSSH -w  
SvcCOPSSH
```

11. Install the Cygwin tools and run the bash shell.

12. Set up the client keys:

```
cp vmquest__id_rsa.pub /cygdrive/c/ssh/home/packt/.ssh/id_rsa.pub
cp vmquest__id_rsa /cygdrive/c/ssh/home/packt/.ssh/id_rsa
chmod 600 /cygdrive/c/ssh/home/packt/.ssh/id_rsa
```

13. Set up the server keys:

```
rm -rf /cygdrive/c/ssh/etc/ssh_host_*
cp vmquest__ssh_host_rsa_key.pub /cygdrive/c/ssh/etc/ssh_host_rsa_key.pub
cp vmquest__ssh_host_rsa_key /cygdrive/c/ssh/etc/ssh_host_rsa_key
chmod 600 /cygdrive/c/ssh/etc/ssh_host_rsa_key
```

14. Set up the host client public key:

```
cp vmhost__id_rsa.pub /cygdrive/c/ssh/home/packt/.ssh/authorized_keys2
```

15. Set up the host key fingerprint:

```
cp vmquest__known_hosts /cygdrive/c/ssh/home/packt/.ssh/known_hosts
```

16. Check whether the connection from host to guest and back works seamlessly:

```
ssh packt@192.168.42.1
```

17. Register the Windows service that will be used to start the build process using the `instsrv` and `srvany` utilities:

```
instsrv.exe packt_build c:\path\to\srvany.exe
reg add "HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\packt_build\Parameters" /v Application /t reg_sz /d "C:\packt\build.bat"
```

18. Configure the service to avoid starting automatically on OS boot:

```
sc config obf_build start= demand
```

19. Complete the manual build of OpenJDK using the *Building OpenJDK 8 on Windows 7 SP1* recipe from *Chapter 4, Building OpenJDK 8*.

How it works...

In this recipe, we created a virtual machine instance with Windows and configured the SSH server to enable seamless automated connections to it.

Binaries of Copssh SSH server Version 3.1.4 were released by the authors as free software under the terms of GNU General Public License Version 3. This means that we can publish or use unchanged binaries for any purposes without additional licensing limitations.

Copssh uses the Cygwin environment and the OpenSSH server under the hood. It also provides integration with the Windows user rights system.

Authorization roles given to the SvcCOPSSH user are required to support the SSH key authentication.

We used Cygwin to set up key files to support the proper setting of Cygwin file rights.

Copssh uses parts of the old version of the Cygwin environment and we need the additional full Cygwin installation for the OpenJDK build process. Different Cygwin versions running on the same machine might interfere with each other causing errors. Although during heavy use of such a setup, I never observed any problems, it is better to keep this point in mind for instances of cryptic Cygwin/Copssh errors/crashes.

Two additional Windows users for the SSH server (sshd and SvcCOPSSH) are used internally by Copssh.

The `ntrights` utility was used to assign additional roles to the SvcCOPSSH user. This utility is not officially supported in the newer versions of Windows but should work fine anyway.

Windows service registration will be required for automated builds to start the actual build process over an SSH connection. For a proper environment setup, the build process on Windows should be started from the `cmd.exe` shell (usually running a batch file). It cannot be started directly from the SSH session that works in the guest Windows machine inside the Cygwin environment. The `instsrv` and `ntrights` utilities allowed us to create a Windows service that will run the batch file (that in turn will start the actual build process) on a path preconfigured in the registry. This `packt_build` service can be started from the SSH session using the `net start` command, effectively starting the build process.

The Manual build was done on this VM to ensure that the environment setup was correct.

There's more...

Other SSH servers can be used in theory, although I am not aware of other free (as in "free speech") SSH server implementations for Windows, which support key authentication.

Other protocols/tools can be used for interaction between the host and guest machine instead of OpenSSH.

See also

- ▶ The *Building OpenJDK 8 on Windows 7 SP1* recipe from Chapter 4, *Building OpenJDK 8*
- ▶ The *Installing Cygwin for Windows builds* recipe from Chapter 2, *Building OpenJDK 6*
- ▶ The *Installing VirtualBox* recipe
- ▶ The *Preparing SSH keys* recipe

- ▶ The Oracle VirtualBox user manual at <https://www.virtualbox.org/manual/UserManual.html>
- ▶ The Copssh website at <https://www.itefix.net/copssh>

Automating builds

This recipe joins together all the previous recipes in this chapter. Prepared virtual machine images and SSH with key authentication will allow us to build OpenJDK in fully automated mode using simple bash scripts without additional tools.

Getting ready

For this recipe, we will require a Linux or Mac OS host machine running.

How to do it...

The following procedure will help us to prepare the Windows virtual machine:

1. Prepare SSH keys as described in the *Preparing SSH keys* recipe in this chapter.
2. Set up the VirtualBox installation and its network settings as described in the *Installing VirtualBox* recipe in this chapter.
3. Prepare the virtual machine images as described in the previous recipes in this chapter.
4. For each VM image, prepare a list of environment variables, which will be used by the build script (for example, Windows):

```
export VM_ADDRESS=192.168.42.1
export VM_NAME=jdk7-windows-amd64
export VM_OSTYPE=Windows7_64
export VM_MEMORY=1780
export VM_IOAPIC=on
export VM_NICTYPE=82545EM
export VM_MACADDR=auto
export VM_OBF_DIR=/cygdrive/c/packt
export VM_START_BUILD="net start obf_build >> build.log 2>&1"
export VM_SHUTDOWN="shutdown /L /T:00 /C /Y"
export VM_IDE_CONTROLLER=PIIX4
```

5. Add the snippets from the following steps (steps 6 to 12) to the main build script.

6. Create a virtual machine instance using the VBoxManage utility:

```
SCRIPT_DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"
VBoxManage createvm --name "$VM_NAME" --register --basefolder
"$SCRIPT_DIR"/target >> "$SCRIPT_DIR"/build.log 2>&1
VBoxManage modifyvm "$VM_NAME" --ostype "$VM_OSTYPE" >> "$SCRIPT_
DIR"/build.log 2>&1
VBoxManage modifyvm "$VM_NAME" --memory "$VM_MEMORY" >> "$SCRIPT_
DIR"/build.log 2>&1
VBoxManage modifyvm "$VM_NAME" --nic1 bridged --bridgeadapter1
tap0 >> "$SCRIPT_DIR"/build.log 2>&1
VBoxManage modifyvm "$VM_NAME" --nictype1 "$VM_NICTYPE" >>
"$SCRIPT_DIR"/build.log 2>&1
VBoxManage modifyvm "$VM_NAME" --macaddress1 "$VM_MACADDR" >>
"$SCRIPT_DIR"/build.log 2>&1
VBoxManage modifyvm "$VM_NAME" --cpus 1 >> "$SCRIPT_DIR"/build.log
2>&1
VBoxManage modifyvm "$VM_NAME" --audio none >> "$SCRIPT_DIR"/
build.log 2>&1
VBoxManage modifyvm "$VM_NAME" --usb off >> "$SCRIPT_DIR"/build.
log 2>&1
VBoxManage modifyvm "$VM_NAME" --vrde on
VBoxManage modifyvm "$VM_NAME" --ioapic "$VM_IOAPIC" >> "$SCRIPT_
DIR"/build.log 2>&1
VBoxManage modifyvm "$VM_NAME" --mouse usbttablet >> "$SCRIPT_DIR"/
build.log 2>&1
VBoxManage modifyvm "$VM_NAME" --keyboard usb >> "$SCRIPT_DIR"/
build.log 2>&1
VBoxManage setextradata global GUI/SuppressMessages
remindAboutAutoCapture,remindAboutMouseIntegrationOn,
showRuntimeError.warning.HostAudioNotResponding,
remindAboutGoingSeamless,remindAboutInputCapture,
remindAboutGoingFullscreen,
remindAboutMouseIntegrationOff,confirmGoingSeamless,
confirmInputCapture,remindAboutPausedVMInput,
confirmVMReset,confirmGoingFullscreen,
remindAboutWrongColorDepth >> "$SCRIPT_DIR"/build.log 2>&1
VBoxManage storagectl "$VM_NAME" --name "IDE" --add ide >>
"$SCRIPT_DIR"/build.log 2>&1
VBoxManage internalcommands sethduuid "$SCRIPT_DIR"/target/$VM_
NAME.vdi >> "$SCRIPT_DIR"/build.log 2>&1
```



```
VBoxManage storageattach "$VM_NAME" --storagectl "IDE" --port 0
--device 0 --type hdd --medium "$SCRIPT_DIR"/target/"$VM_NAME".vdi
>> "$SCRIPT_DIR"/build.log 2>&1
```

```
VBoxManage storagectl "$VM_NAME" --name "IDE" --controller "$VM_
IDE_CONTROLLER"
```

7. Start up the virtual machine instance:

```
VBoxManage startvm "$VM_NAME" --type headless >> "$SCRIPT_DIR"/
build.log 2>&1
```

```
ssh "$VM_ADDRESS" "ls" > /dev/null 2>&1
while [ $? -ne 0 ]; do
echo "Waiting for VM ..."
    sleep 10
    ssh "$VM_ADDRESS" "ls" > /dev/null 2>&1
done
echo "VM started"
```

8. Enable remote logging back to the host machine over SSH:

```
ssh "$VM_ADDRESS" "cd "$VM_OBF_DIR" && echo 'Starting build' >
build.log"
nohup ssh "$VM_ADDRESS" "tail -f "$VM_OBF_DIR"/build.log | ssh
192.168.42.2 'cat >> "$SCRIPT_DIR"/build.log'" >> "$SCRIPT_DIR"/
build.log 2>&1 &
LOGGER_PID="$!"
```

9. Copy the OpenJDK sources into the build VM and start the build:

```
scp "$SCRIPT_DIR"/openjdk.zip "$VM_ADDRESS":"$VM_OBF_DIR"
ssh "$VM_ADDRESS" "cd "$VM_OBF_DIR" && unzip -q openjdk.zip >>
build.log 2>&1"
ssh "$VM_ADDRESS" "cd "$VM_OBF_DIR" && "$VM_START_BUILD"
```

10. Poll the build machine periodically looking for the `build_finished.flag` file that should be created after the build is finished:

```
ssh "$VM_ADDRESS" "if [ ! -f "$VM_OBF_DIR"/build_finished.flag ];
then exit 1; else exit 0; fi" > /dev/null 2>&1
while [ $? -ne 0 ]; do
    echo "Waiting for build ..."
    sleep 300
ssh "$VM_ADDRESS" "if [ ! -f "$VM_OBF_DIR"/build_finished.flag ];
then exit 1; else exit 0; fi" > /dev/null 2>&1
done
```

11. Copy the build results, stop the logger, shut down the virtual machine instance, and unregister it from VirtualBox:

```
scp -r "$VM_ADDRESS":"$VM_OBF_DIR"/dist/* "$SCRIPT_DIR"/dist
kill -9 $LOGGER_PID >> "$SCRIPT_DIR"/build.log 2>&1
ssh "$VM_ADDRESS" "$VM_SHUTDOWN" >> "$SCRIPT_DIR"/build.log 2>&1
|| true
sleep 15
VBoxManage controlvm "$VM_NAME" poweroff > /dev/null 2>&1 || true
VBoxManage unregistervm "$VM_NAME" >> "$SCRIPT_DIR"/build.log
```

12. To start the build with the chosen virtual machine image, use the following commands:

```
. windows_amd64.env # please not the dot before the command
nohup build.sh >> build.log 2>&1 &
echo "$!" > .pid
tail -F "$SCRIPT_DIR"/build.log
```

13. After the build is finished, the OpenJDK binaries will be copied back to the host machine.

How it works...

We use the low-level `VBoxManage` tool to manipulate virtual machine instances for better control over this process.

To ensure that the virtual machine instance actually starts, we poll it periodically over SSH and stop polling after the first successful connection.

For remote logging, we run the `tail -f` process on the build machine that sends its output back to the host machine immediately over SSH. We start this process with a connection from the host machine in the background using the `nohup` utility and write the host process `pid` to the `.pid` file to kill the process after the build.

We use `scp` to copy sources to a build machine and commands over SSH to decompress the sources and start the build.

After the build is started, we poll the build machine periodically over SSH to look for the `build_finished.flag` file that should be created by the build script on the build machine.

After the build is finished, we copy OpenJDK binaries back to the host machine and shut down the virtual machine instance gracefully before unregistering it.

Different virtual machine configuration and environment options are listed in the `.env` file for each virtual machine image. We use the "dot first" syntax to import variables from the `env` file to the current shell. This allows us to use a generic build script for all virtual machine images.

There's more...

This recipe can be seen as a basic example of build automation. Different commands, tools, and protocols can be used to achieve the same goal.

The `build_finished.flag` file (with custom content) can also be used to end the build prematurely after the error.

See also

- ▶ The *Installing VirtualBox* recipe
- ▶ The *Preparing SSH keys* recipe
- ▶ The *Preparing VirtualBox machines with Linux* recipe
- ▶ The *Preparing VirtualBox machines with Mac OS X* recipe
- ▶ The *Preparing VirtualBox machines with Windows* recipe
- ▶ *Chapter 4, Building OpenJDK 8*
- ▶ The Oracle VirtualBox user manual at <https://www.virtualbox.org/manual/UserManual.html>

Building cross-platform installers

When cloud services became the ubiquitous way to install desktop software, classic GUI installers became almost obsolete. Cloud package repositories or stores can be much more convenient to install and, in the first place, update the desktop application.

At the same time, GUI installers are still widely used for various free and commercial applications. Especially for cross-platform applications, GUI installers should show the same behavior on all supported platforms despite not being fully native on those platforms. Some applications require complex environment changes at the time of installation, for example, registering themselves as Windows services or setting environment variables.

In this recipe, we will prepare a cross-platform installer for OpenJDK that will work on all supported platforms (Windows, Linux, and Mac OS X). We will use a popular open-source installation tool, `IzPack`, written in Java.

Getting ready

For this recipe we will require the OpenJDK binaries (to wrap into the installer).

How to do it...

The following procedure will help us in preparing the installer:

1. Download IzPack compiler Version 4.3.5 from the IzPack website (<http://izpack.org/>) and install it.
2. Download the sample installation config file from the documentation section of the Izpack website.
3. Move the `jre` directory from the OpenJDK image one level up, next to the `openjdk` directory.
4. Add the `jre` directory to the installer configuration as a "loose" pack using the following configuration snippet:

```
<pack name="OpenJDK RE" required="yes" loose="true">
  <description>OpenJDK Runtime Environment</description>
  <file src="jre" targetdir="$INSTALL_PATH"/>
</pack>
```

5. Add the `openjdk` directory (that now does not contain JRE) as a normal pack:

```
<pack name="OpenJDK DK" required="no">
  <description>OpenJDK Development Kit</description>
  <fileset dir="openjdk" targetdir="$INSTALL_PATH"/>
  <file src="uninstall" targetdir="$INSTALL_PATH"/>
</pack>
```

6. Adjust labels, GUI forms, locale, and icons as you like.
7. Run the IzPack compiler:


```
./izcomp/bin/compile ./config.xml -h ./izcomp -o installer.jar
```
8. Put the generated `install.jar` and `jre` directory in the `openjdk-installer` directory.
9. Add the bash/batch script to the `openjdk-installer` directory which will allow us to run the installer using the relative path to the `jre` directory:


```
./jre/bin/java -jar install.jar
```
10. Compress the `openjdk-installer` directory—it now contains the OpenJDK installer.

How it works...

The main feature of our installer is that the installer itself runs on the same version of Java that it will install. The `loose="true"` configuration in the JRE pack instructs the installer to find this pack on a relative path outside of the main installation `.jar` file, without duplicating the contents of the `jre` directory.

There's more...

The IzPack installer supports a lot of configuration options, we highlighted only the basic one in this recipe. Besides the GUI and installation forms customizations, the one feature that can be useful, especially for OpenJDK, is running scripts at installation time. We can prepare scripts to adjust the environment and add them to the corresponding packs using the executable configuration element. On simple Unix-like operating systems, such scripts can simply append the `PATH` variable changes to `~/.bashrc` or `~/.bash_profile` files. On Windows, utilities such as `pathman`, `setx`, and `reg` can be used to adjust the environment variables or the Windows Registry.

Instead of running scripts at the time of installation, you can extend the IzPack itself (adding new forms, and so on) and perform environment registration directly from Java code.

See also

- ▶ *Chapter 2, Building OpenJDK 6*
- ▶ *Chapter 3, Building OpenJDK 7*
- ▶ *Chapter 4, Building OpenJDK 8*
- ▶ The IzPack installer website at <http://izpack.org/>

Index

Symbols

32-bit FreeType libraries

- building, for OpenJDK 7 on Windows 52, 53
- building, for OpenJDK 6 on Windows 34, 35

32-bit OpenJDK 6

- building, on Windows 7 SP1 38-40

32-bit OpenJDK 7

- building, on Windows 7 SP1 55-57

64-bit FreeType libraries

- building, for OpenJDK 7 on Windows 54
- building, for OpenJDK 6 on Windows 36, 38

64-bit OpenJDK 6

- building, on Windows 7 x64 SP1 42, 43

64-bit OpenJDK 7

- building, on Windows 7 x64 SP1 59, 60

@bug tag 177

@compile tag 178

@run tag 178

@summary tag 177

@test tag 177

A

abstract syntax tree (AST) 243

AdoptOpenJDK project

- about 162
- benefits 162
- javadocs, securing 163, 164
- JIT logs, visualizing 162, 163

amd64 24, 46

Ant

- URL 170

architectures

- building, OpenEmbedded
- recipes used 105-107

Autoconf. *See* GNU Autoconf

B

benchmark project

- creating, JMH used 193-200

Betterrev 228

bootstrap builds 80

build automation

- performing 270-274

build-dep command 71, 82, 84

bylaws

- URL 205

C

C++ code

- debugging, NetBeans used 133-135

CA certificates

- preparing 24-26
- URL, for list 26

ccache tool

- about 72
- used, for speeding OpenJDK 8
- build process 72

Certificate Authority certificates. *See* CA certificates

chmod 777 command 58

Clang

- used, for building OpenJDK 9 on
- Mac OS X 233, 234

code review

- creating 228-230

contributor

- becoming 206, 207

cross-compilation, between ARM and x86

- configuring 92, 93

cross-platform installers

- building 274-276

custom test

writing, for jtreg 175-177

Cygwin

installing, for Windows builds 32, 33

D

debug command 136

defect

submitting, to OpenJDK
bug systems 224-226

development environment

setting up, with NetBeans 120-125

dev options

AdaptiveSizePolicyReadyThreshold 138
DieOnSafepointTimeout 137
EagerInitialization 138
FullGCALot 138
GuaranteedSafepointInterval 138
InlineUnsafeOps 137
MaxTrivialSize 138
MemProfiling 137
MinInliningThreshold 138
ScavengeWithObjectsInToSpace 137
SharedOptimizeColdStartPolicy 138
UsePrivilegedStack 137
Verbose 137
VerifyParallelOldWithMarkSweep 137
ZapJNIHandleArea 137
ZapResourceArea 137
ZapUnusedHeapArea 137

dev parameters, HotSpot

using 136, 137

F

fastdebug command 136

file local.properties

bytecodelib 184
jcommjar 184
jhalljar 184
jhjar 184
junitlib 184
servletjar 184

FreeType

about 35, 52
building 35
URL 35, 52

FT_CONFIG_OPTION_SUBPIXEL_RENDERING

macro 53

FT_EXPORT_DEF macro 53

FT_EXPORT macro 53

G

General Availability status 45

GNU Autoconf

about 67
working with 67, 68

GNU Compiler for Java (GCJ) 80

governance structure, Governing Board

Chair role 205
OpenJDK Lead role 205
two At-Large members 205
Vice Chair role 205

Graal

used, for building OpenJDK 9 243-245

H

Hob utility

about 95
used, for configuring OpenEmbedded
build 95-98

HotSpot

compiling, NetBeans used 135, 136
dev parameters, using 136, 137
intrinsic, adding 138-144

I

i586 term 24, 46

IcedTea

about 79, 80
building for ARM, with integrated
CACAO VM 100, 101

icedtea-2* source code tarball

URL 83

IcedTea 6

building 80-82
building, with SystemTap support 88, 89

icedtea6* source code tarball

URL 81

IcedTea 7

building 83, 84
building, with NSS security provider 87, 88

IcedTea Java WebStart implementation

preparing, for Mac OS X 115, 116
preparing, for Windows 117, 118
using, on Linux 113, 114

IcedTea patches

OpenJDK 7, building with 85, 86

IcedTea-Web browser plugin

about 110
building, on Linux 110-112

IcedTea-Web project 109**intrinsic**

about 138
adding, to HotSpot 138-144

J**JamVM**

about 102
porting 102, 103

Java Class Library (JCL) 7**Java code**

debugging, NetBeans used 129-133

Java Community Process (JCP) 214**Java Compatibility Kit (JCK) 165****Java Concurrency Stress Test. *See* jcstress****Java KeyStore (JKS) format 25****Java Microbenchmark Harness. *See* JMH****Java Money API**

URL 232

Java Network Launching Protocol (JNLP)

syntax 113

Java Specification Request. *See* JSRs**JavaTest 165****Java Test Harness. *See* JT Harness****Java WebStart technology 109****jcstress**

about 186
building 186, 187
running 188
tests, writing for 189-193

JDK9

project page, URL 22

JDK bug tracking system

URL 174

JDK Lead 203**JEPs**

about 216

suggesting 216-219

JEPs, Jigsaw

JEP 200 240
JEP 201 240
JEP 220 240

JHM

compiling 201

Jigsaw

about 239, 242
building 240, 241
module, installing as library 242
using 239

JIRA

about 223
URL 223

JIT log visualizer 164**Jitwatch**

about 162
executable jar, URL 162

JMH

about 193
used, for creating benchmark
project 193-200

JMX Group webpage

URL 212

JSRs

about 113
suggesting 214-216

JT Harness

about 183
compiling, from source code 183-185
URL 170

JT_JAVA 168**jtreg**

about 165
building, from source 169-171
custom test, writing for 175-177
TestNG tests, writing for 180-182
URL 167
used, for running tests 167-169
using, in GUI mode 178-180

jtreg command line arguments

-agentvm 169
-automatic 169
-jdk 168
-r 169
-verbose:summary 169

-w 169
URL 168

JUnit

URL 170

L

libmotif-dev package 82

Linux

IcedTea browser plugin, building on 110-112
IcedTea Java WebStart implementation,
using 113, 114
OpenJDK, configuring 18-21
OpenJDK, installing 16-18
VirtualBox machines, preparing with 262-264

LogCompilation parameter 163

M

Mac OS X

IcedTea Java WebStart implementation,
preparing for 115, 116
OpenJDK 7, building 49-51
OpenJDK 8, building 73-75
OpenJDK 9, building 233, 234
VirtualBox machines,
preparing with 264, 265

Mercurial

about 125
installing 125
URL 125
working with 125-127

minimum build environment

setting up 30, 31

MIPS

building, OpenEmbedded recipes used 105

MSYS

about 234
used, for building OpenJDK 9 on
Windows 234-236

N

NetBeans

used, for compiling HotSpot 135, 136
used, for creating patch 226, 227
used, for debugging C++ code 133-135
used, for debugging Java code 129-133

used, for setting up development
environment 120-125

NetBeans All bundle

URL 120

NETwork eXecute (NetX)

about 110, 113
module, building 113

Network Security Services (NSS) 87

NetX next.jar file 116

NSS security provider

IcedTea 7, building with 87, 88

ntrighs utility 269

O

OpenEmbedded

about 92, 93
architectures, adding 106
build, configuring 94
build, configuring with Hob utility 95-98
compile problems 100
configure problems 99
fetch problems 99
parsing problems 100
software, adding 106
using 93
working 98

OpenJDK

about 7, 8
configuring, on Linux 18-21
configuring, on Windows 14-16
defect, submitting to bug systems 224-226
distinguishing, from Oracle JDK 9-11
groups 21, 22, 211, 212
installing, from unofficial builds 12
installing, on Linux 16-18
installing, on Windows 11-14
projects 21, 22, 213
URL 21

OpenJDK 6

about 23, 24
building, on Ubuntu Linux 12.04 LTS 27-30
family tree 24
incremental builds 127, 128

OpenJDK 7

32-bit FreeType libraries, building on
Windows 52, 53

- 64-bit FreeType libraries, building on
 - Windows 54
- about 45, 46
- building, on Mac OS X 49-51
- building, on Ubuntu Linux 12.04 LTS 46-48
- building, with IcedTea patches 85, 86
- incremental builds 127, 128

OpenJDK 8

- about 66
- build environment, configuring 67, 68
- build process, speeding with ccache 72
- building, on Mac OS X 73-75
- building, on Windows 7 SP1 75-78
- versus OpenJDK 9 232

OpenJDK 8 Ubuntu Linux 12.04 LTS

- building 69-71

OpenJDK 9

- building, from source 239
- building on Mac OS X, Clang used 233, 234
- building on Windows, MSYS used 234-237
- building, with Graal 243-245
- building, with Sumatra 245-256
- early access preview, running 237, 238
- early access preview, testing 238, 239
- versus OpenJDK 8 232

OpenJDK community

- about 203
- contributor, becoming 206, 207

OpenJDK v9 patches

- backporting, to OpenJDK v8 209, 211

OpenSSH 259

Oracle Contributor Agreement (OCA)

- about 204
- URL 206

Oracle JDK

- OpenJDK, distinguishing from 9-11

Oracle VirtualBox user manual

- URL 259

P

patch

- creating, NetBeans used 226, 227
- generating, with webrev 207, 209

pause > nul command 58, 78

plugin, VisualVM

- creating 148-161

PrintAssembly parameter 163

process workflow

- navigating through 222-224

project author 204

public-key cryptography 24

R

Reference Implementation (RI) 215

S

Secure Shell (SSH) protocol 259

SetEnv.Cmd script 58

Shark 104

source code, JHM

- building 201
- downloading 200

ssh-keygen command 262

SSH keys

- preparing 259-262
- vmguest__id_rsa 261
- vmguest__id_rsa.pub 261
- vmguest__known_hosts 261
- vmguest__ssh_host_rsa_key 261
- vmguest__ssh_host_rsa_key.pub 261
- vmhost__id_rsa 261
- vmhost__id_rsa.pub 261
- vmhost__known_hosts 261
- vmhost__ssh_host_rsa_key 261
- vmhost__ssh_host_rsa_key.pub 261

standalone toolchain, for 32- and 64-bit

Windows' builds

- preparing 61-64

standard tests, OpenJDK

- about 172
- running 172-174

Sumatra

- about 253
- used, for building OpenJDK 9 245-248

system calls (syscalls) 88

SystemTap support

- IcedTea 6, building with 88, 89

T

tail -f command 58, 78

Technology Compatibility Kit (TCK) 216

tee command 58

TestNG tests

writing, for jtreg 180-182

TestNG v.6.8

URL 170

tests

running, jtreg used 167, 168

TraceClassLoading parameter 163

Transport Layer Security (TLS) 87

troubleshooting

code review, creating 228-230

defect, submitting to OpenJDK bug
systems 224-226

patch, creating with NetBeans 226, 227

process workflow, navigating
through 222-224

Truffle 243

U

Ubuntu

URL 263

Ubuntu Linux 12.04 LTS

OpenJDK 6, building on 27-30

OpenJDK 7, building on 46-48

UnlockDiagnosticVMOptions parameter 163

V

VirtualBox

about 258

installing 258, 259

machines, preparing with Linux 262-264

machines, preparing with Mac OS X 264, 265

machines, preparing with Windows 266-268

Visual Studio 2005 Express Edition

about 38

URL 36

VisualVM

building, from source code 146, 147

plugin, creating 148-161

W

webrev

used, for generating patch 207-209

Windows

32-bit FreeType libraries,

building for OpenJDK 6 34, 35

32-bit FreeType libraries,

building for OpenJDK 7 52, 53

64-bit FreeType libraries,

building for OpenJDK 6 36, 37

64-bit FreeType libraries,

building for OpenJDK 7 54

IcedTea Java WebStart implementation,

preparing for 117, 118

OpenJDK 9, building 234-236

OpenJDK, configuring 14-16

OpenJDK, installing 11-14

VirtualBox machines, preparing with 266-268

Windows 7 SP1

32-bit OpenJDK 6, building 38-41

32-bit OpenJDK 7, building 55-57

OpenJDK 8, building 75-78

Windows 7 x64 SP1

64-bit OpenJDK 6, building 42, 43

64-bit OpenJDK 7, building 59, 60

Windows builds

Cygwin, installing 32, 33

Windows Server 2003 SP1 Platform SDK

URL 37

wslauncher.sh script 116

X

Xalan

URL 171

Xcode 5

URL 233

Z

Zero-assembler HotSpot port

about 103

configuring, with Shark compiler 103-105



About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



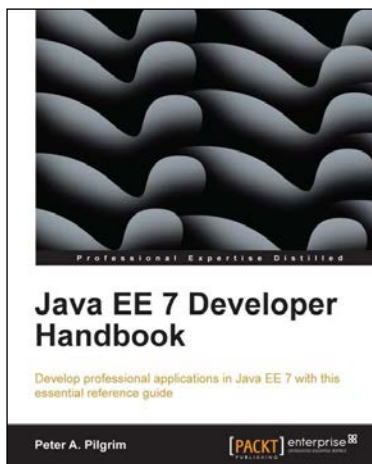
Instant Web Scraping with Java

ISBN: 978-1-84969-688-3

Paperback: 72 pages

Build simple scrapers or vast armies of Java-based bots to untangle and capture the Web

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results.
2. Get your Java environment up and running.
3. Gather clean, formatted web data into your own database.
4. Learn how to work around crawler-resistant websites and legally subvert security measures.



Java EE 7 Developer Handbook

ISBN: 978-1-84968-794-2

Paperback: 634 pages

Develop professional applications in Java EE 7 with this essential reference guide

1. Learn about local and remote service endpoints, containers, architecture, synchronous and asynchronous invocations, and remote communications in a concise reference.
2. Understand the architecture of the Java EE platform and then apply the new Java EE 7 enhancements to benefit your own business-critical applications.
3. Learn about integration test development on Java EE with the Arquillian framework and the Gradle build system.

Please check www.PacktPub.com for information on our titles



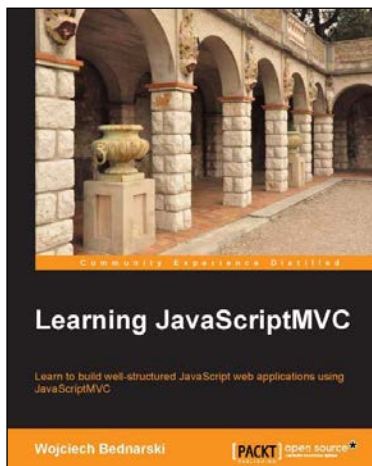
Java EE 7 with GlassFish 4 Application Server

ISBN: 978-1-78217-688-6

Paperback: 348 pages

A practical guide to install and configure the GlassFish 4 application server and develop Java EE 7 applications to be deployed to this server

1. Install and configure GlassFish 4.
2. Covers all major Java EE 7 APIs and includes new additions such as JSON Processing.
3. Packed with clear, step-by-step instructions, practical examples, and straightforward explanations.



Learning JavaScriptMVC

ISBN: 978-1-78216-020-5

Paperback: 124 pages

Learn to build well-structured JavaScript web applications using JavaScriptMVC

1. Install JavaScriptMVC in three different ways, including installation using Vagrant and Chef.
2. Document your JavaScript codebase and generate searchable API documentation.
3. Test your codebase and application as well as learn how to integrate tests with the continuous integration tool—Jenkins.

Please check www.PacktPub.com for information on our titles

