

01

Getting started with Groovy

In this chapter, we will cover:

- Installing **Groovy** on **Windows**
- Installing **Groovy** on **Linux** and **OSX**
- Executing **Groovy** code from the command-line
- Using **Groovy** as a command-line text file editor
- Using **Groovy** to start server on the command-line
- Running **Groovy** with **invokodynamic** support
- Building **Groovy** from source
- Managing multiple **Groovy** installations on **Linux**
- Using **groovysh** to try out **Groovy** commands
- Starting **groovyConsole** to execute **Groovy** snippets
- Configuring **Groovy** in **Eclipse**
- Configuring **Groovy** in **IntelliJ IDEA**

Introduction

This first chapter focuses on the basics of getting started with **Groovy**. We start by showing how to install **Groovy** on the most popular operating systems and we move to some command line tools available with the language distribution. The remaining recipes offer an overview of how the language easily integrates with the most popular **Java IDEs**.

Installing Groovy on Windows

In this recipe we will provide instructions on installing the **Groovy** distribution on **Windows** operating system.

Getting ready...

The requirement for **Groovy 2.0** installation is **JDK 1.5+**. We assume that the reader has a **JDK** installed and knows how to use **Java**. In case you use **JDK 7** or later, then you can take advantage of dynamic language optimization present in that version i.e. the **invokedynamic** byte code instruction (see *"Running Groovy with **invokedynamic** support"* recipe).

To install **Groovy** on **Windows** you need to download the **ZIP** distribution from <http://groovy.codehaus.org/Download>.

The latest major version at the time of writing is **2.1.6**. The latest minor version is **2.0.8**. Since version **2.0**, **Groovy** has changed the release version numbering, so that the next major version of **Groovy** will be **3.0** and the next minor versions will have the second digit increased (**2.1**, **2.2**, **2.3** and so on).

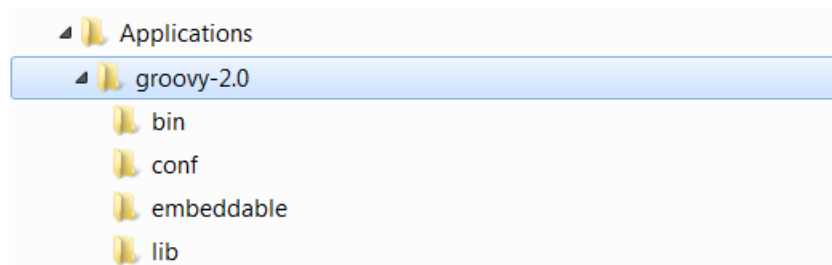
Alternatively you can build **Groovy** from the source distribution which is described in *"Building Groovy from source"* recipe.

How to do it...

After downloading the zipped distribution, you need to unzip the archive to a directory of your choice.

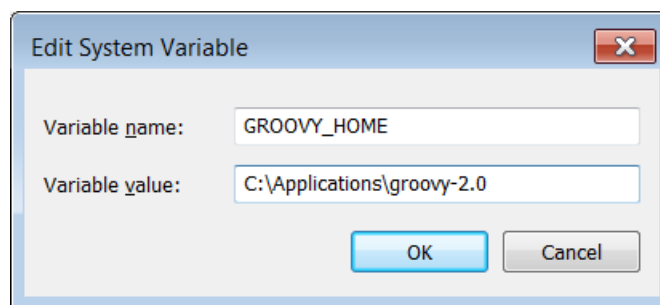
1. For simplicity we will assume that it is **C:\Applications\groovy-2.0**.

The contents of the directory should look like the following:



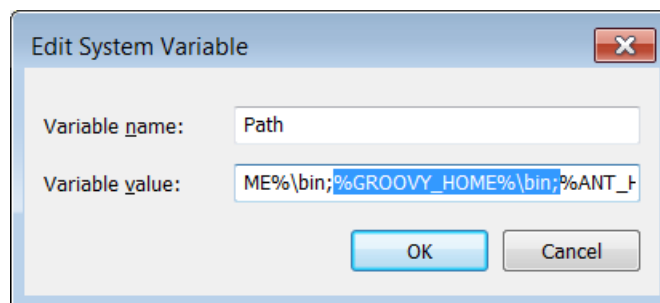
9366OS_01_22.png

2. In order to get the **groovy** command available on your command line you need to add it to your system's path by setting the environment variable named **PATH**. We also advice to create a **GROOVY_HOME** variable for simpler reference.
3. To access the **Windows** environment variables, you need to press the **Windows+Break** key combination. On **Windows Vista/7** or later it will open the "Control Panel" page for system settings.
4. Click on the "Advanced System Settings" to open the "System Properties" window.
5. Then you need to click the "Environment Variables..." button to finally get to the list of system variables.
6. Click the "New..." button and add the **GROOVY_HOME** variable pointing to your **Groovy** installation path:



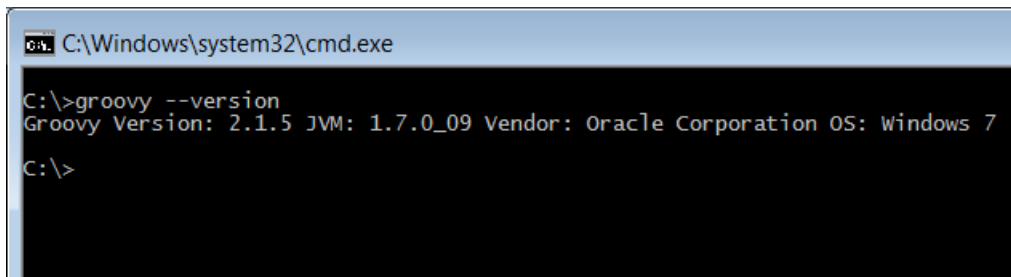
9366OS_01_23.png

7. Then find the **PATH** variable in the list of system variables and append or insert the **%GROOVY_HOME%\bin;** string to it:



9366OS_01_24.png

8. You can now fire the **Windows** command-line and verify that **Groovy** is installed correctly by issuing the **groovy --version** command:



```
C:\Windows\system32\cmd.exe

C:\>groovy --version
Groovy Version: 2.1.5 JVM: 1.7.0_09 Vendor: Oracle Corporation OS: Windows 7

C:\>
```

9366OS_01_25.png

If you get the output displayed in the image above, your **Groovy** installation is complete.

There's more...

As an alternative to the zipped archive, **Windows** users can also download a one-click installer (you can find the link on the same download page under "[Download Windows-Installer](#)" link). Simply run it and follow the instructions to get a fully functional **Groovy** installation.

Installing Groovy on Linux and OSX

This recipe gives instructions for installing **Groovy** on any **Linux** distribution and **Mac OSX**.

Getting ready...

As a starter, download the **Groovy 2.0** binaries as described in recipe "*Installing Groovy on Windows*".

How to do it...

1. Create a new folder for the **Groovy** distribution:
| **sudo mkdir /usr/share/groovy**
2. Move the unzipped **Groovy** folder into **/usr/share/groovy** and create a symlink to the folder, without using the version number.
| **sudo mv groovy-2.1.6 /usr/share/groovy/**

- | **sudo ln -s /usr/share/groovy/groovy-2.1.6 current**
- 3. Finally add **Groovy** to the path by editing your `~/.profile` (or `~/.bash_profile`) file. You can use **vi** or an editor of your choice.
 - | **export GROOVY_HOME=/usr/share/groovy/current**
 - | **export PATH=\$GROOVY_HOME/bin:\$PATH**
- 4. Your **JAVA_HOME** variable should be set as well. On **OSX** the recommended way is to use:
 - | **export JAVA_HOME=\$(/usr/libexec/java_home)**
- 5. Reload your `~/.profile` file by typing:
 - | **source ~/.profile**
- 6. To test if your installation is successful, just type:
 - | **groovy -version**

The output should display the installed **Groovy** version and the **JDK** in use.

How it works...

Using a symbolic link called **current**, which we created in step 2, makes it very easy to upgrade to newer version of **Groovy** by changing the folder the symbolic link points to.

There's more...

Most ***nix**-based operating systems (like **Linux** or **Mac OSX**) have package manager systems that allow to install **Groovy** by simply typing a command in the terminal.

In **Ubuntu**, **Groovy** can be installed by simply typing:

```
| sudo apt-get install groovy
```

The version installed by the **Ubuntu** package manager is quite old (1.7.10) so, you may want to install **Groovy** manually as described in this recipe.

In **OSX** you can use **Homebrew**:

```
| brew install groovy
```

If you are happy with running a stable version of **Groovy** - but likely not the most recent one, a package manager is the recommended way to get **Groovy** quickly and easily. If you want to install a beta version of **Groovy** or a version that is not yet available on the package manager system used by your **OS**, install the binaries from the website.

See also

If you are in need of supporting several **Groovy** distributions in your **Linux** environment you can also take a look at *"Managing multiple Groovy installations on Linux"* recipe.

Executing Groovy code from the command-line

Groovy is by definition a language with "scripting features": many developers approach **Groovy** by writing short scripts to automate repetitive tasks. The language provides a set of command-line tools that help you to create scripts that are usable within your favorite shell.

In this recipe we will cover the execution of a simple script with the help of **groovy** command, which is made available to you after a successful **Groovy** installation (see *"Installing Groovy on Windows"* and *"Installing Groovy on Linux and OSX"* recipes).

How to do it...

Let's start with the most abused example in programming books, printing "Hello, World!".

1. The simplest way to execute **Groovy** code is by using the **-e** option and start writing **Groovy** code in the same line:
| **groovy -e "println 'Hello, world!'"**
2. You can also place the **println 'Hello, world!'** statement in a separate file e.g. **hello.groovy** and execute that script with the following simple command:
| **groovy hello.groovy**
3. In both cases you'll see the same result:
| **Hello, world!**

How it works...

In step 1, the actual **Groovy** code resides in the double quotes (") and uses the predefined **println** method to print a "Hello, World!" string. But to explain where actually **println** method comes from, we need to give a bit more details about **Groovy** internals.

Every script in **Groovy** (a command-line parameter or a standalone script file) is compiled on the fly into a class that extends **groovy.lang.Script** class (**Javadoc** for that can be found at <http://groovy.codehaus.org/api/groovy/lang/Script.html>).

Naturally, a **Script** class is eventually inherited from **java.lang.Object**, which is the base class for all classes both in **Java** and **Groovy**. But since **Groovy** adds its own extension methods to many standard **JDK** classes (see *"Adding functionality to existing Java/Groovy classes"* recipe for information on custom extension modules), **java.lang.Object** is enriched with many useful methods including **println** (relevant

Javadoc can be found at [http://groovy.codehaus.org/groovy-jdk/java/lang/Object.html#println\(java.lang.Object\)](http://groovy.codehaus.org/groovy-jdk/java/lang/Object.html#println(java.lang.Object))).

There's more...

In fact, **groovy** command has several other useful command line options. If you type **groovy --help**, you can get a full list of them:

```
c:\>groovy --help
usage: groovy [options] [args]
options:
  -a,--autosplit <splitPattern>    split lines using splitPattern <default '\s'>
                                     using implicit 'split' variable
  -c,--encoding <charset>           specify the encoding of the files
  -classpath <path>                  Specify where to find the class files - must
                                     be first argument
  -cp,--classpath <path>            Aliases for '-classpath'
  -d,--debug                         debug mode will print out full stack traces
  -D,--define <name=value>          define a system property
  --disableopt <optlist>            disables one or all optimization elements.
                                     optlist can be a comma separated list with
                                     the elements: all <disables all
                                     optimizations>, int <disable any int based
                                     optimizations>
  -e <script>                       specify a command line script
  -h,--help                         usage information
  -i <extension>                   modify files in place; create backup if
                                     extension is given (e.g. '.bak')
  -l <port>                          listen on a port and process inbound lines
                                     <default: 1960>
  -n                                process files line by line using implicit
                                     'line' variable
  -p                                process files line by line and print result
                                     <see also -n>
  -v,--version                      display the Groovy and JVM versions
```

9366OS_01_01.png

Let's go through some of those options to get a better overview of possibilities.

First of all, **-classpath**, **--classpath** and **-cp** options work in a very similar way to the **java** command. You just specify a list of ***.jar** files or list of directories with ***.class** files. The only peculiarity is that **-classpath** must come as the first parameter in the command line, otherwise **Groovy** will not recognize that.

Another parameter that is common with **java** command is **-D**, and that allows to pass system properties to your script in the following way:

```
groovy -Dmessage=world
-e "println 'Hello, ' + System.getProperty('message')"
```

One of the **Groovy** strengths (as opposed to **Java**) is conciseness. This rule is also applied to what **Groovy** prints out in case exception occurs in your script:

```
groovy -e "throw new Exception()"
Caught: java.lang.Exception
java.lang.Exception
```

```
| at script_from_command_line.run(script_from_command_line:1)
To print the conventional full Java stack trace you can use -d or -debug options (some
stack trace lines are omitted for brevity):
groovy -d -e "throw new Exception()"
Caught: java.lang.Exception
java.lang.Exception
    at sun.reflect.Native...
    ...
    at script_from_command_line.run(script_from_command_line:1)
    ...
    at org.codehaus.groovy.tools.GroovyStarter.main(...)
```

See also

For additional command line features please examine the following recipes:

- *"Using Groovy as a command-line text file editor" recipe*
- *"Using Groovy to start server on the command-line" recipe*

For more information of **Groovy** script structure and **Groovy** additions look at:

- <http://groovy.codehaus.org/api/groovy/lang/Script.html>
- <http://groovy.codehaus.org/groovy-jdk/java/lang/Object.html>

Using Groovy as a command-line text file editor

The **groovy** command, which we introduced in the *"Executing Groovy code from the command-line"* recipe, can also be used as a stream editor or text file filter. In this recipe we will cover the **-i**, **-n** and **-p** parameters that can be used to leverage file editing and processing functionality.

How to do it...

Assume that you have a file, **data.txt**, that contains 5 lines with numbers from **1** to **5**.

1. For example, to multiply each number by 2, you can use the following command:

```
| groovy -n -e "println line.toLong() * 2" data.txt
```

2. We can even omit the **println** method call if we pass additional **-p** parameter to the command:

```
| groovy -n -p -e "line.toLong() * 2" data.txt
```

3. In both cases, **Groovy** will print the following:

```
| 2
| 4
```



```
| 6  
| 8  
| 10
```

How it works...

Due to the fact we are using the `-n` option, the code in double quotes is applied to each line read from a data file that is specified as the last parameter in the command line.

The `line` variable is predefined by **Groovy**, and you can use it to access, filter or modify line's content.

There's more...

If you add the `-i` option to the command above, then it will actually modify the input file with output values:

```
| groovy -i -n -p -e "line.toLong() * 2" data.txt
```

Adding a suffix `.bak` to `-i` option will save the original input file `data.txt` under `data.txt.bak`:

```
| groovy -i .bak -n -p -e "line.toLong() * 2" data.txt
```

You can use `-n` and `-p` options to also filter the input stream of other operating system commands. For example, to filter directory listing (`dir`) command output to show only `*.jar` files on **Windows** you can use the following command:

```
| dir | groovy -n -e "if (line.contains('.jar')) println line"
```

Or on ***nix**-based operating systems:

```
| ls -la | groovy -n -e "if (line.contains('.jar')) println line"
```

Of course, the result of the above commands can be easily achieved by more efficient operating system instructions. But these examples are given to demonstrate that you can actually leverage the full power of the **Groovy** and **Java** programming languages to implement more complex processing rules.

See also

For more information on using **Groovy**'s command-line interface you can check the following recipes:

- ["Executing Groovy code from the command-line" recipe](#)
- ["Using Groovy to start server on the command-line" recipe](#)

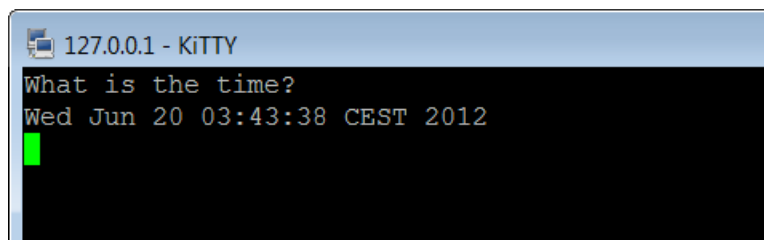
Using Groovy to start server on the command-line

In this recipe we continue to explore the **groovy** command's features at one's disposal. This time we show how to create a process capable of serving client requests through **TCP/IP** directly from the command line and with one line of code.

How to do it...

The command-line option that we are going to use for this purpose is **-l**.

1. By using that option, it's trivial to start a simple socket server in **Groovy**:
| **groovy -l 4444 -e "println new Date()"**
2. The above line will start a server that listens to port **4444** and returns the date and time string for every line of data it receives from the clients:
| **groovy is listening on port 4444**
3. In order to test that the server actually works, you can start any **telnet**-like program (e.g. **KiTTY** if you are on **Windows**) to connect to **localhost** on port **4444**, and type any string (e.g. "What time is it?") and hit **Enter** in the end. Server should reply with a date/time string back:



A screenshot of a terminal window titled "127.0.0.1 - KiTTY". The terminal shows a prompt "What is the time?" followed by the response "Wed Jun 20 03:43:38 CEST 2012". A green cursor is visible on the line following the response.

9366OS_01_02.png

In this way you can quite easily organize communication channels for *ad hoc* notifications on different hosts.

See also

For more information on using **Groovy**'s command-line interface you can check the following recipes:

- *"Executing Groovy code from the command-line" recipe*
- *"Using Groovy as a command-line text file editor" recipe*

Running Groovy with invokedynamic support

One of the biggest improvements introduced in **Groovy 2.0** is the support for the **invokedynamic**. The **invokedynamic** is a new **JVM** instruction available in **Java 7** that allows easier implementation and promises increased speed and efficiency of dynamic languages (like **Groovy**).

The bytecode generated by dynamic languages tends to require several actual **JVM** method invocations in order to perform one in the dynamic language. There is extensive use of reflection and dynamic proxies, which have a performance cost. Also the **JIT**, *Just-In-Time compiler*, that helps to improve runtime performance of a **JVM**, can not work its magic by applying optimization to the bytecode because it lacks information and patterns which is normally possible to optimize. The new bytecode instruction, **invokedynamic**, is able to partially mitigate these issues, including support for better **JIT** optimization.

In this recipe we will show how to run **Groovy** with the **invokedynamic** (also known as *indy*) support and **Java 7**.

Getting ready...

The **invokedynamic** support is a compile-time and runtime feature only. In other words, a developer can not use it from within the source code. What **invokedynamic** brings to **Groovy 2.0** (and even more **2.1**) is basically improved runtime performance.

How to do it...

As explained in the introduction, **Java 7** is the required **JVM** to compile **Groovy** code that leverages the **invokedynamic** instruction.

1. Make sure that **Java 7** is your current **JVM**. Type `java -version` and confirm that the output mentions version 7:

```
java version "1.7.0_25"
...
```
2. The following steps will let us fully enable *indy* support in your **Groovy** distribution. First of all rename or remove all the JAR files starting with **groovy-** in the **lib** directory of your **Groovy 2.x** home directory.
3. Replace them with the files located in the **indy** directory located in the root of the **Groovy** distribution folder.
4. Remove the **-indy** classifier from the jar names.

5. Finally, invoke either **groovy** or the **groovyc** compiler with the **--indy** flag to execute your code:

```
| groovy --indy my_script.groovy
```

There's more...

It is important to note that if the **--indy** flag is omitted, the code will be compiled **without** **invokedynamic** support, even if **Java 7** is used and the **Groovy** JAR files have been replaced.

The performance gain introduced by the new **JVM** instruction greatly varies depending on a numbers of factors, including the actual **JVM** version and the type of code that is optimized. **JVM** support for the **invokedynamic** instruction improves at each version: the upcoming **Java 8** will use **invokedynamic** for supporting *lambda* functions so it is very likely that newer **JVMs** will offer even greater optimization. Given the current state of things, some benchmarks that we have run have shown an improvement of around 20% given the same code compiled with the **invokedynamic** enabled.

See also

You can also check **Groovy**'s "*Getting Started Guide*" for more information on the **invokedynamic** support (<http://groovy.codehaus.org/InvokeDynamic+support>).

Building Groovy from source

In this recipe we introduce a procedure for building **Groovy** from the source code. The only requirement needed to build **Groovy** from sources is **Java JDK 1.7** or higher.

Java 7 is required to leverage the new **invokedynamic** instruction used by **Groovy 2**. You can read more about the benefits of **invokedynamic** in the "*Running Groovy with invokedynamic support*" recipe.

Getting ready...

As many of today's open source projects, **Groovy** source is maintained on **GitHub**. **GitHub** is a website providing **Git** hosting service. You probably have heard of **Git**, the version control system started by the **Linux** creator, Linus Torvalds.

In order to build **Groovy**, you need a local copy of the source code that must be fetched from **GitHub** via **Git**. If you are running a **Linux** or **OSX** operating system, chances are that you have already **Git** installed on your box.

On **Windows** there are several ways to install **Git**. You may want to use **Cygwin** (<http://www.cygwin.com/>) or the officially released version available on the **Git** website (<http://git-scm.com/download/win>).

For this recipe, we assume that a recent version of **Git** is available in your shell. To test that **Git** is indeed available, open a shell and type:

```
| git --version
```

How to do it...

Assuming that **git** is installed and operational we can proceed with the following steps.

1. Open a shell in your operating system and type:
| **git clone https://github.com/groovy/groovy-core.git**
2. Wait for **Git** to fetch all the source code and proceed to build **Groovy**. On **Windows**, open a **DOS** shell, move to the **groovy-core** folder you just cloned and type:
| **gradlew.bat clean dist**
On **Linux** or **Mac OSX**, open a shell, move to the **groovy-core** folder and type:
| **./gradlew clean dist**
3. If you already have **Gradle** installed you can run this command instead:
| **gradle clean dist**

How it works...

The **git clone** command in the first step fetches the **Groovy** repository - around **125MB**, so be patient if you are on a slow connection. **Groovy** has switched to the **Gradle** build tool from **Ant**: the **gradlew** command is a convenient wrapper for **Gradle** that takes care of downloading all the required dependencies and trigger the build. The "02. Using Groovy ecosystem" chapter has a whole recipe dedicated to **Gradle**, so you may want to take a look at the "Integrating Groovy into build process using Gradle" recipe to know more about this awesome tool. Furthermore, several recipes of this book will make use of **Gradle** to build the code examples.

The build process will download the required dependencies and compile the code. Upon successful compilation, the build will generate a **ZIP** file named **groovy-binary-2.x.x-SNAPSHOT.zip** containing the binaries, under **/target/distributions**. Install the binaries in the same way as explained in the "Installing Groovy on Windows" and "Installing Groovy on Linux and OSX" recipes.

Note how two types of **Groovy** binaries are generated: a normal version and an **indy**-version. The **indy** version will leverage the **invokedynamic** feature (see also "Running Groovy with **invokedynamic** support" recipe).

Managing multiple Groovy installations on Linux

If a developer needs to work with different **Groovy** distributions on the same machine, chances are that he or she would be involved in a lot of environment variable fiddling - **PATH**, **JAVA_HOME** and **GROOVY_HOME**.

Luckily there is a tool that helps to manage those variables as well as to download the required setup files on demand.

The name of this goody is **GVM** - **Groovy enVironment Manager**. **GVM** was inspired by similar tools from the **Ruby** ecosystem, **RVM** and **rbenv**.

In this recipe, we will demonstrate how to use the **GVM** tool and show the benefits it delivers.

Getting ready...

Use the package manager available on your **Linux** distribution to install **curl**, **unzip** and **java** on your machine. For example, on **Ubuntu** it can be achieved with the following command sequence:

```
| sudo apt-get update
| sudo apt-get install curl
| sudo apt-get install zip
| sudo apt-get install openjdk-6-jdk
```

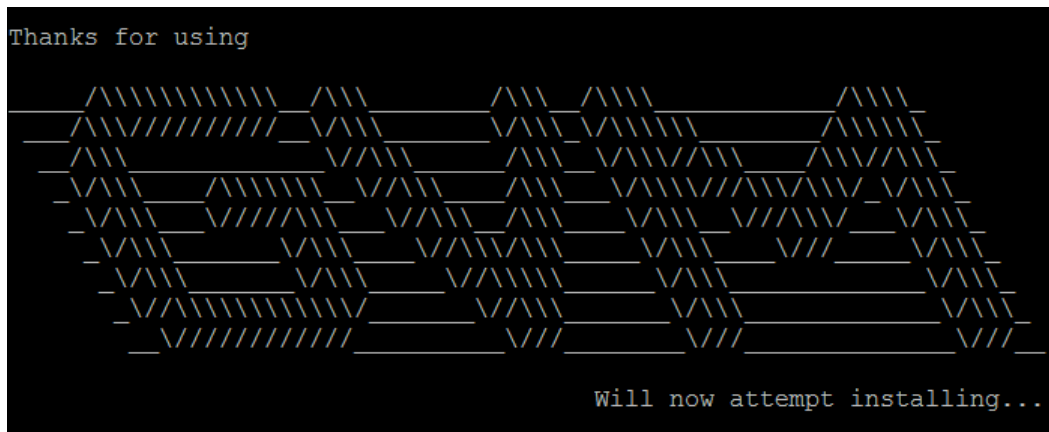
The **GVM** installation script will not work without those packages. One additional step that you have to perform before you can run the installer is to create the **GVM**'s cache directory in the current user's home folder:

```
| mkdir -p ~/.gvm/var
```

Then you need to fetch the installation script from **GVM**'s website (<http://get.gvmtool.net>) and pass it to **bash** using the following command:

```
| curl -s get.gvmtool.net | bash
```

It will start the set up process:



9366OS_01_26.png

To finalize the installation, run this command:

```
| source "~/gvm/bin/gvm-init.sh"
```

How to do it...

As soon as **GVM** is installed and running you can start putting it to use.

1. To install the latest **Groovy** distribution you can issue the following command:

```
|> gvm install groovy
Downloading: groovy 2.1.6
...
Installing: groovy 2.1.6
Done installing!
```

2. At the end of the installation it will ask you whether to make it default or not. Since we all like the latest and greatest, type **Y** (yes):

```
|Do you want groovy 2.1.6 to be set as default? (Y/n): Y
```

3. To install different **Groovy** distribution (e.g. **1.8.6** which is still rather popular) you can fire the following:

```
|> gvm install groovy 1.8.6
Downloading: groovy 1.8.6
...
Installing: groovy 1.8.6
Done installing!
```

4. Again it will ask about setting **1.8.6** as default **Groovy** distribution. Answer **n** (no) in this case, since we would like to keep version **2** as the primary one:

```
|Do you want groovy 1.8.6 to be set as default? (Y/n): n
```

5. To set (or to ensure) **Groovy** version used by default do this:

- ```
> gvm default groovy 2.1.6
Default groovy version set to 2.1.6
```
6. You can also verify that **Groovy** is running and is of requested version by typing:
- ```
> groovy --version
Groovy Version: 2.1.6 JVM: ...
```
7. To temporarily switch to a different **Groovy** distribution just type:
- ```
> gvm use groovy 1.8.6
Using groovy version 1.8.6 in this shell.
```
8. Another way to check, which **Groovy** is active now, is:
- ```
> gvm current groovy
Using groovy version 1.8.6
```
9. For example, this script will not run under 1.8.6:
- ```
> groovy -e "println new File('.').directorySize()"
Caught: groovy.lang.MissingMethodException:
 No signature of method:
 java.io.File.directorySize() is applicable ...
```
10. If we switch to the latest **Groovy**, the script will succeed:
- ```
> gvm use groovy
Using groovy version 2.1.6 in this shell.
> groovy -e "println new File('.').directorySize()"
126818311
```
11. To remove unneeded distribution we can just run:
- ```
> gvm uninstall groovy 1.8.6
Uninstalling groovy 1.8.6...
```

## How it works...

The reason the `directorySize()` method (steps 9 and 10) didn't work for version **1.8.6** of **Groovy** is simply because that method was only introduced in version **2**.

As we already mentioned, **GVM** manages the values of environment variables to direct your **Groovy** commands to the proper distribution. It also downloads, unpacks and caches **Groovy** installation archives under `~/.gvm/var` directory.

## There's more...

**GVM** also can manage other popular **Groovy**-based products like **Gradle** (build framework that we are going to discuss in *"Integrating Groovy into build process using Gradle"*), **Grails** (web application framework), **Griffon** (desktop application framework) etc. in a similar way.

This recipe can be applied also to **Mac** running **OSX**. You can enjoy **GVM** on **Windows** too, but you need to install and run **Cygwin** (**Linux** environment simulation for **Windows**).



## See also

Additional useful information can be found on the following product home pages:

- **GVM** home page - <http://gvmtool.net/>
- **Cygwin** home page - <http://www.cygwin.com/>

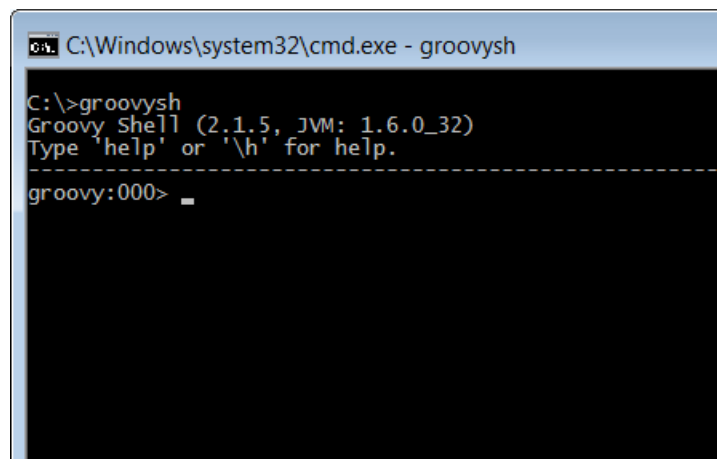
## Using groovysh to try out Groovy commands

Similarly to many other languages (e.g. **Ruby** or **Perl**), **Groovy** sports a so called "Read - Evaluate - Print loop" (**REPL**). **REPL** is a simple, interactive programming environment mostly used to quickly try out language features. Groovy's **REPL** is named **groovysh** and in this recipe we are going to explore some of its feature.

## How to do it...

**groovysh** is a command line tool available with the standard **Groovy** distribution. Install **Groovy** as described in one of the installation recipes (see "*Installing Groovy on Windows*" and "*Installing Groovy on Linux and OSX*") and you'll get the **groovysh** command available in your shell.

1. Open a shell on your operating system and type **groovysh**:



```
C:\Windows\system32\cmd.exe - groovysh
C:\>groovysh
Groovy Shell (2.1.5, JVM: 1.6.0_32)
Type 'help' or '\h' for help.

groovy:000> _
```

9366OS\_01\_27.png

2. The **Groovy** shell allows you to evaluate simple expressions like:

```
groovy:000> println "Hello world!"
Hello world
```

```
| ==> null
```

3. It is also possible to evaluate more complex expressions like functions or closures (closures are discussed in great length in the *"Defining code as data in Groovy"* recipe):

```
groovy:000> helloClosure = { println "Hello $it" }
==> groovysh_evaluate$_run_closure1@7301061
groovy:000> counter = 1..5
==> 1..5
groovy:000> counter.each helloClosure
Hello 1
Hello 2
Hello 3
Hello 4
Hello 5
==> 1..5
```

4. The **Groovy** shell supports also creating classes:

```
groovy:000> class Vehicle {
groovy:001> String brand
groovy:002> String type
groovy:003> String engineType
groovy:004> }
==> true
groovy:000> v = new Vehicle()
==> Vehicle@7639fabd
groovy:000> v.brand = 'Ferrari'
==> Ferrari
groovy:000> v.type = 'Car'
==> Car
==> null
groovy:000> println v.brand
Ferrari
==> null
```

5. The dynamic nature of **Groovy** allows us to quickly list all the methods on a class:

```
groovy:000> GString.methods.each { println it}
public java.util.regex.Pattern groovy.lang.GString.negate()
public boolean groovy.lang.GString.equals(java.lang.Object)
public boolean
groovy.lang.GString.equals(groovy.lang.GString)
public java.lang.String groovy.lang.GString.toString()
public int groovy.lang.GString.hashCode()
...
```

## How it works...

The **groovysh** command compiles and executes completed statements as soon as we press the *Enter* key, and prints the result of that statement execution along with any output from the execution

Autocompletion is supported through the [JLine](http://jline.sourceforge.net/) library

<http://jline.sourceforge.net/>. Pressing the **TAB** key automatically completes keywords and methods as we type.

```
groovy:000> string = "I'm a String!"
==> I'm a String!
groovy:000> string.
Display all 159 possibilities? (y or n)
groovy:000> string.toU
```

```
toURI() toURL() toUpperCase(toUpperCase()
```

In step 2, the evaluated statement returned **null**. This is normal, **groovysh** is informing us that the last statement didn't return any value - hence **null**.

In step 4, we can see how **groovysh** supports code that spawns multiple lines. Note how the counter on the left of each statement increases at each line. The **up** and **down** arrows key will display the history of the typed commands: history is preserved even across sessions, so you can safely exit **groovysh** and you will still be able to access the history.

You may have noticed that in the examples above, we didn't use any typed variables. A variable declared with **def** or a data type is not stored in the session and will be "lost" as soon as the command is issued.

```
groovy:000> def name = "Oscar"
==> Oscar
groovy:000> println name
ERROR groovy.lang.MissingPropertyException:
No such property: name for class: groovysh_evaluate
 at groovysh_evaluate.run (groovysh_evaluate:2)
```

This is a small gotcha you should remember when using **groovysh**.

## There's more...

**groovysh** has a number of commands, information on which can be accessed by typing **help**:

```

groovy:000> help

For information about Groovy, visit:
 http://groovy.codehaus.org

Available commands:
 help (\h) Display this help message
 ? (\?) Alias to: help
 exit (\x) Exit the shell
 quit (\q) Alias to: exit
 import (\i) Import a class into the namespace
 display (\d) Display the current buffer
 clear (\c) Clear the buffer and reset the prompt counter.
 show (\s) Show variables, classes or imports
 inspect (\n) Inspect a variable or the last result with the GUI object browser
 purge (\p) Purge variables, classes, imports or preferences
 edit (\e) Edit the current buffer
 load (\l) Load a file or URL into the buffer
 . (\.) Alias to: load
 save (\s) Save the current buffer to a file
 record (\r) Record the current session to a file
 history (\H) Display, manage and recall edit-line history
 alias (\a) Create an alias
 set (\=) Set (or list) preferences
 register (\rc) Registers a new command with the shell

For help on a specific command type:
 help command

```

## 9366OS\_01\_28.png

The **import** behaves like the standard **import** keyword in **Groovy** and **Java**. It allows to import packages from the **JDK** or the **GDK**.

```

groovy:000> import groovy.swing.SwingBuilder
==> [import groovy.swing.SwingBuilder]
groovy:000> swing = new SwingBuilder()
==> groovy.swing.SwingBuilder@6df59ac1
groovy:000> frame = swing.frame(title:'Frame') {
groovy:000> textlabel = label(text:'hello world!')
groovy:000> }
==> javax.swing.JFrame[...]
groovy:000> frame.show()
==> null

```

The **display** command shows the current buffer and **save** allows to save it to a file.

```

groovy:000> display
Buffer is empty
groovy:000> class Person {
groovy:001> display

```

With `clear` the buffer can be reset, in case you mistyped something.

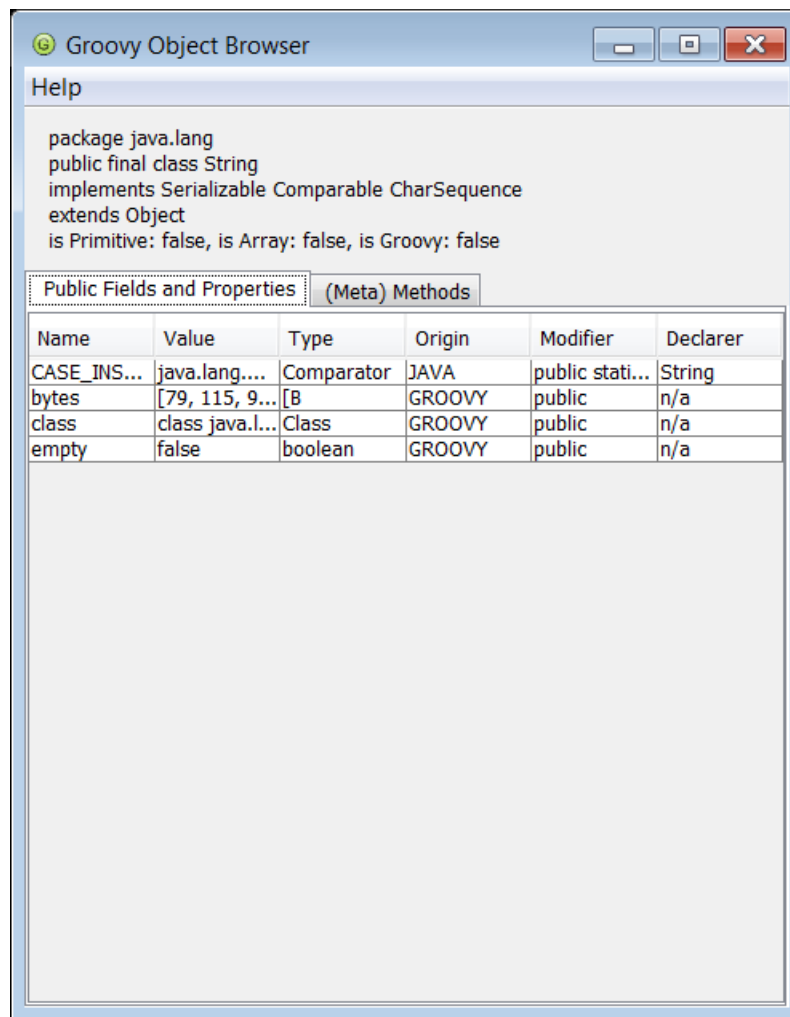
The `record` command act as a flying recorder. It saves the typed commands to a file as we type. Use `record start` and `record stop` to control the recording. It is preferable to specify a file to which you want the recorded commands to be stored.

```
groovy:000> record start Session1.txt
Recording session to: Session1.txt
===> Session1.txt
groovy:000> println 'hello world!'
hello world!
===> null
groovy:000> class Person {
groovy:001> String name
groovy:002> }
===> true
groovy:000> record stop
Recording stopped; session saved as: Session1.txt (202 bytes)
===> Session1.txt
```

A very useful feature of the `Groovy` console is the `inspect` command that displays the content of the last evaluated expression inside a `GUI` application, named "`Groovy Object Browser`".

The "`Groovy Object Browser`" shows a good deal of information about the latest stored object, such as the class name, the implemented interfaces and all the methods exposed by the object. The next image shows some of the methods visible in the `java.lang.String` class.

```
groovy:000> name = "Oscar"
===> Oscar
inspect
```



9366OS\_01\_29.png

## Starting groovyConsole to execute Groovy snippets

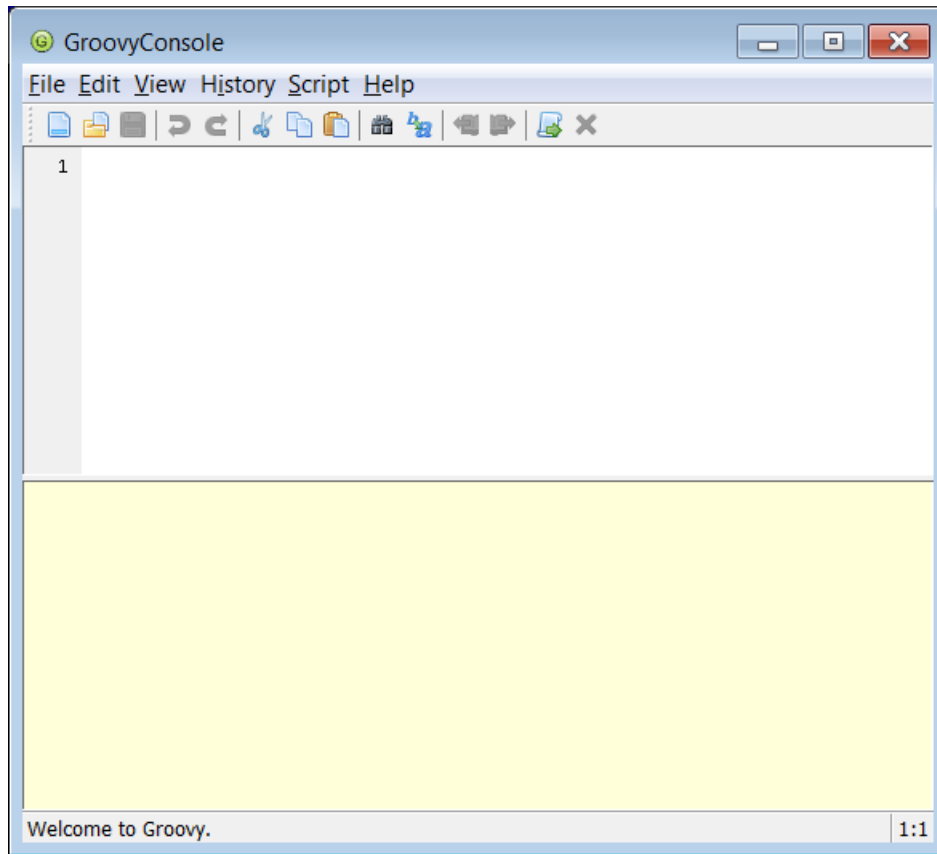
The **Groovy** distribution has another option for trying out code snippets for developers who are more familiar with a graphical interface. The tool in question is **groovyConsole**, which is very similar to **groovysh** (see *"Using groovysh to try out Groovy commands"* recipe) except that it provides a **GUI** with syntax highlighting and basic code editor features.

In this recipe, we are going to cover basic "GroovyConsole" usage scenarios.

## How to do it...

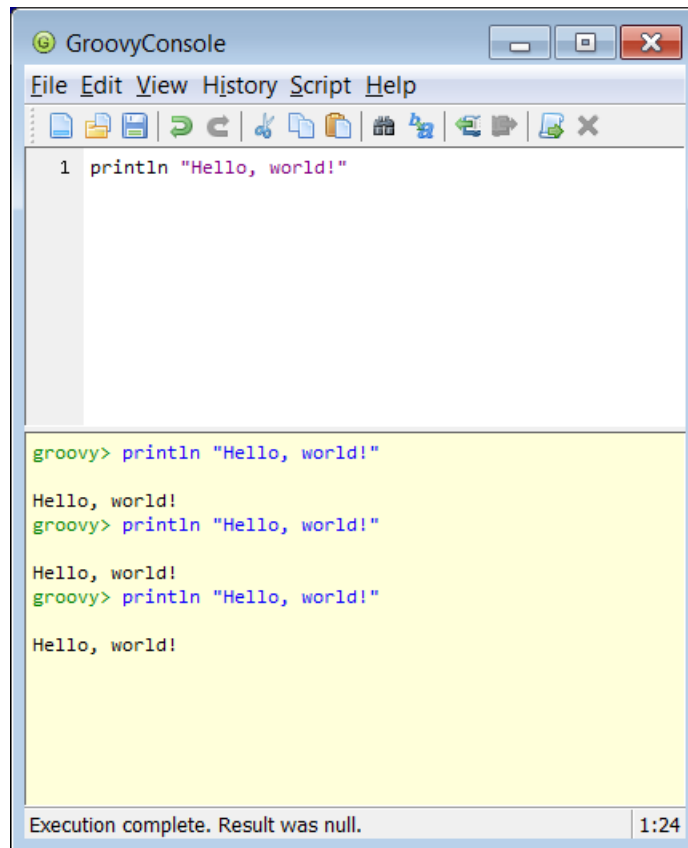
Open your shell and let's dive in.

1. To start "GroovyConsole" you just need to fire the `groovyConsole` command on your command-line:



9366OS\_01\_30.png

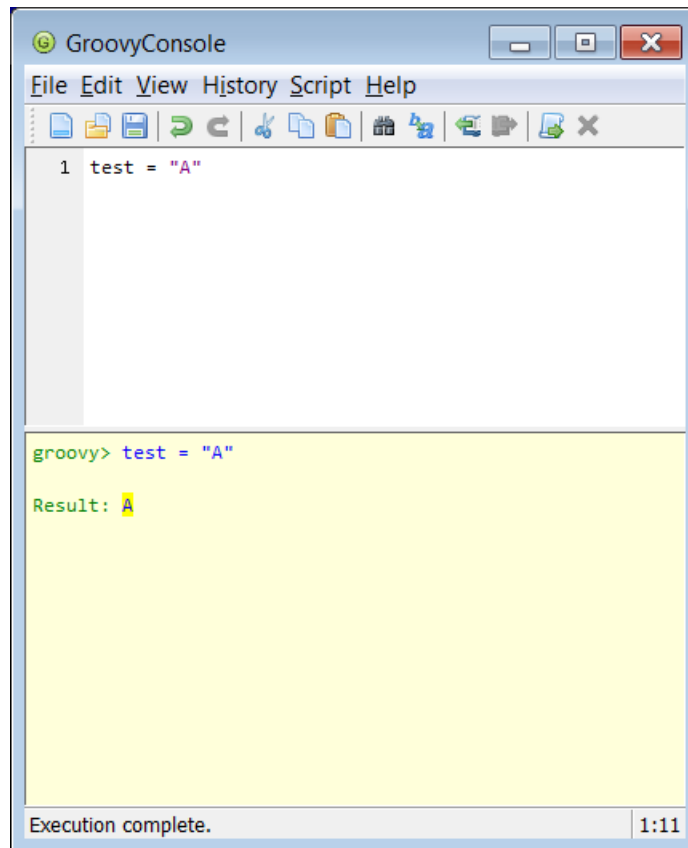
2. "GroovyConsole" screen is divided into 2 logical parts. The top area is meant for script editing and the bottom area displays the script execution output when a script is launched.
3. By pressing `Ctrl+R` (`CMD+R` on a **Mac**) - or going to "Script" menu and selecting "Run" - the script in the edit area will be executed.



### 9366OS\_01\_31.png

4. Press **Ctrl+W** (**CMD+W** on a **Mac**) to clean the output in the lower pane.
5. The bottom panel displays an output similar to the **groovysh** shell i.e. first the executed script is printed after the 'groovy>' marker followed by the printed output and a result value, if any.

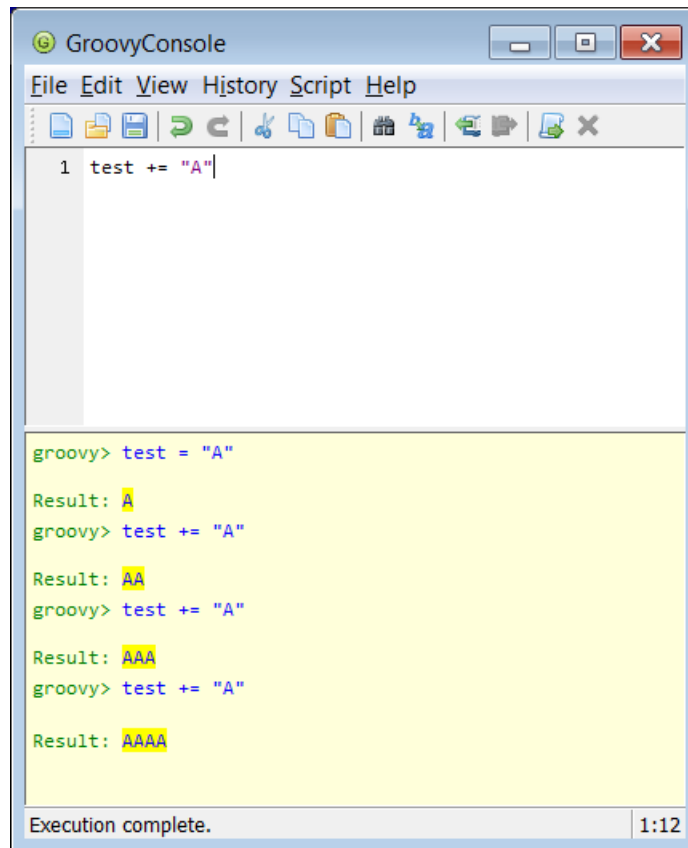




### 9366OS\_01\_32.png

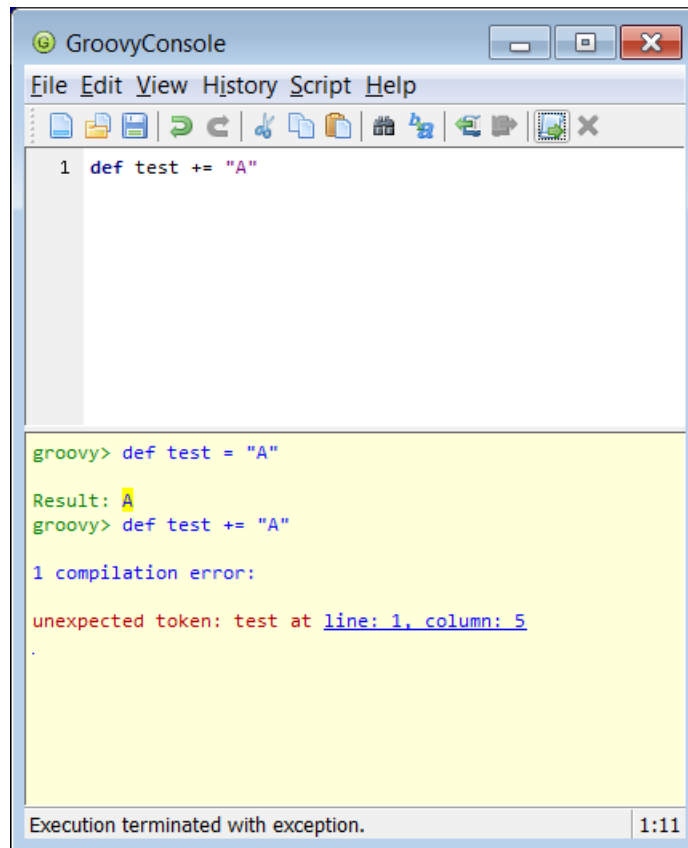
6. In the above script we set the value of the internal `test` property (this property is being stored within an instance of `groovy.lang.Script` class that is used to execute all the code in the console).

If you change that code to reuse the previous field value (e.g. `+=`), you can actually notice that the value changes every time that the code gets executed:



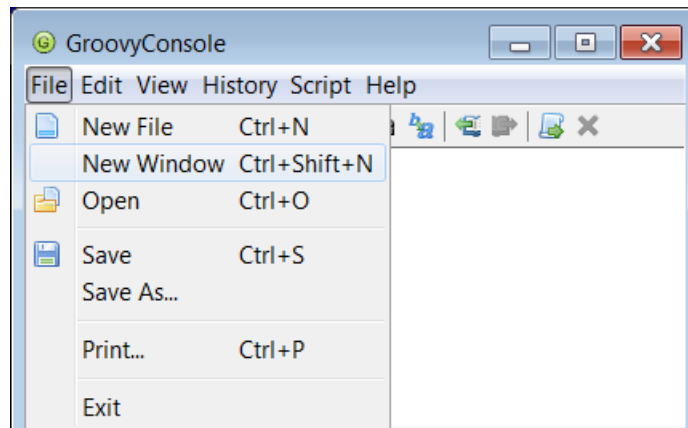
9366OS\_01\_33.png

7. Similarly to `groovysh`, if you define a variable using the `def` keyword, then you can't reuse the previous value. `def` is used to define locally scoped variables and in this case it's bound only to single execution of the script:



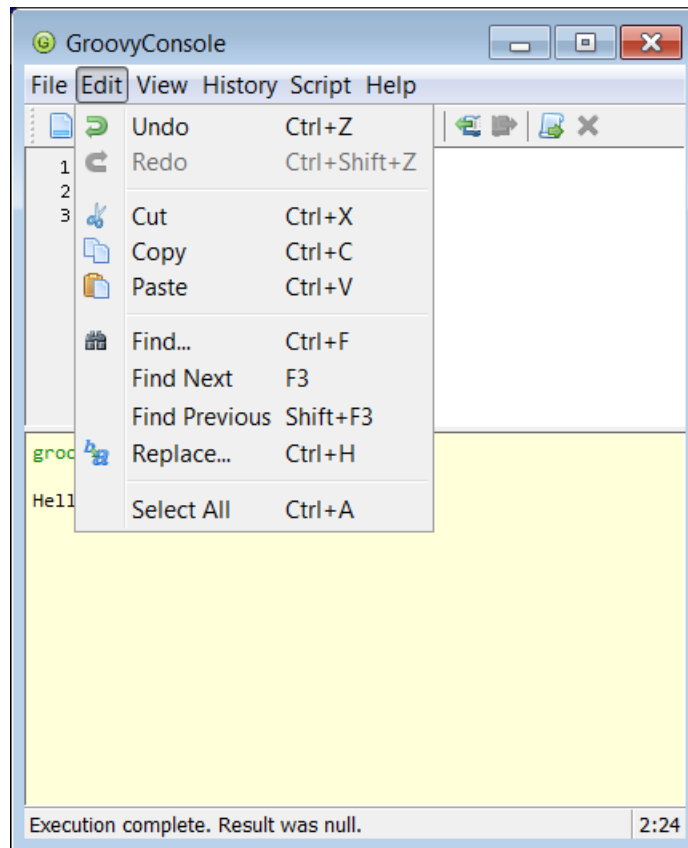
9366OS\_01\_34.png

8. Let's go through the editor features of "GroovyConsole" application. The "File" menu offers standard functions to start over by clicking on the "New File" menu item, to open a new "GroovyConsole" window, to open a previously saved **Groovy** script and to save or print contents of the editor and to close the application.



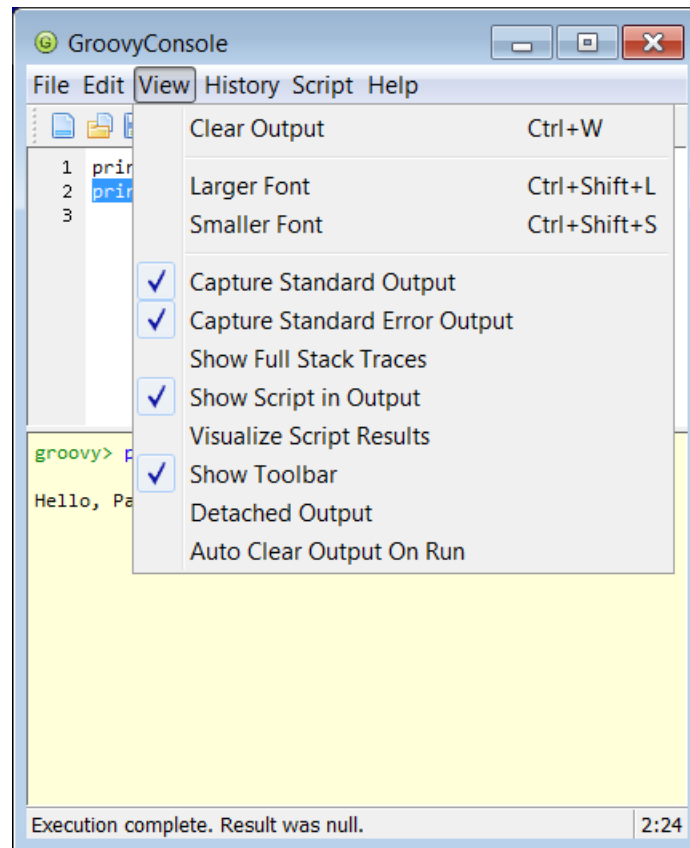
9366OS\_01\_35.png

9. The "Edit" menu provides functionality available in any modern text editor such as copy, paste and basic search functionality:



9366OS\_01\_36.png

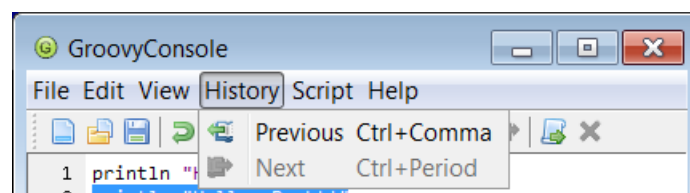
10. The "View" menu has some interesting features for manipulating what is displayed in the bottom output area:



## 9366OS\_01\_37.png

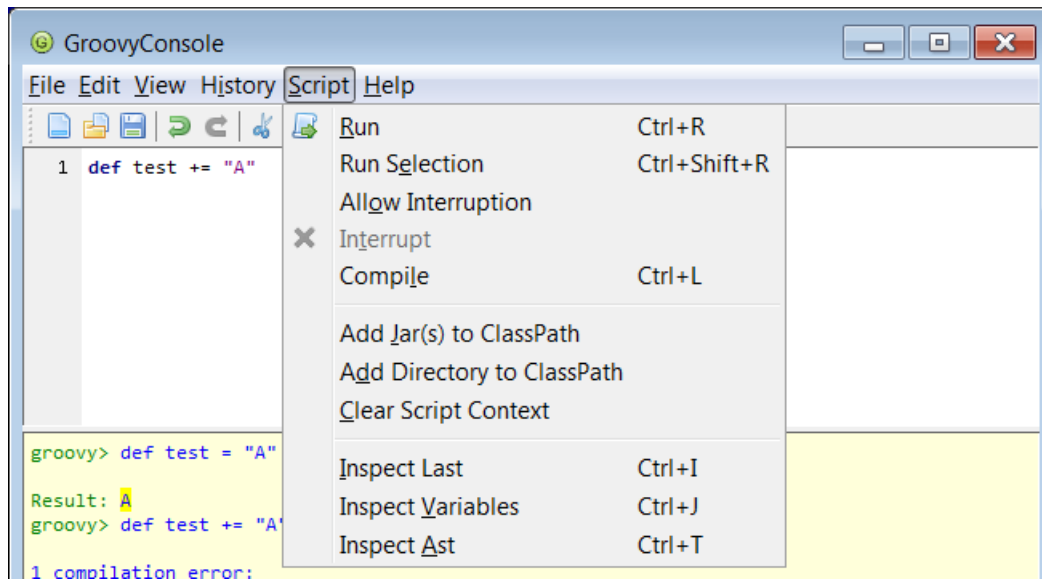
It is possible to increase or decrease the font size and configure the way the output panel display information. For instance, you can disable the script output or clear the output screen at each run.

11. The "History" menu allows you to revive previous script versions that you have executed during your session with "GroovyConsole":



## 9366OS\_01\_38.png

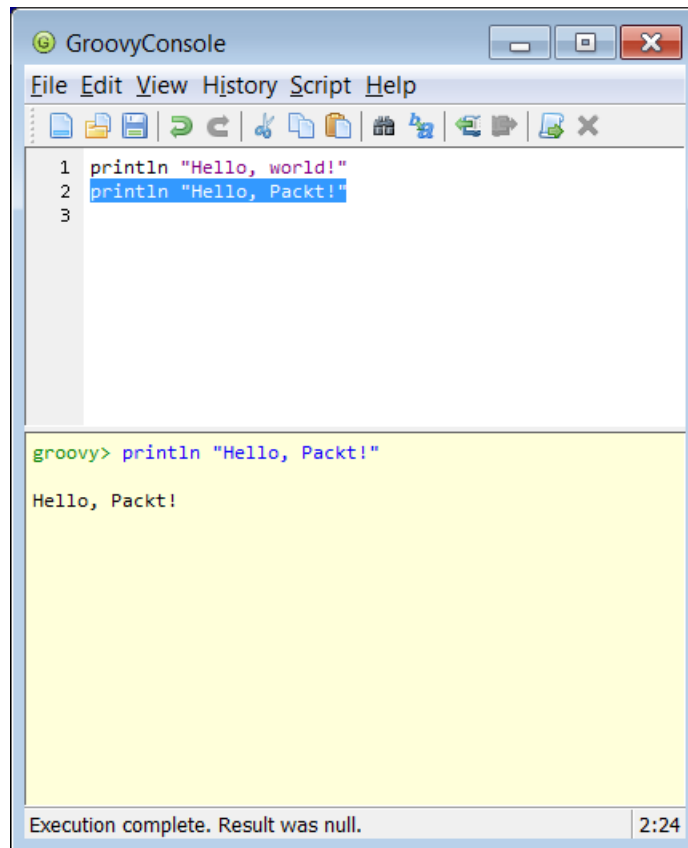
12. Since "GroovyConsole" is all about executing a script, the "Script" menu presents the most valuable options:



## 9366OS\_01\_39.png

You can run or abort your script execution, force script compilation, add additional libraries to the class path. Also you can "Clear Script Context", which will lead to clearing all the script's accumulated properties (e.g. `test`).

13. Another useful feature is that you can select part of your script in the editor and execute only that by clicking "Run Selection" in "Script" menu:



9366OS\_01\_40.png

## There's more...

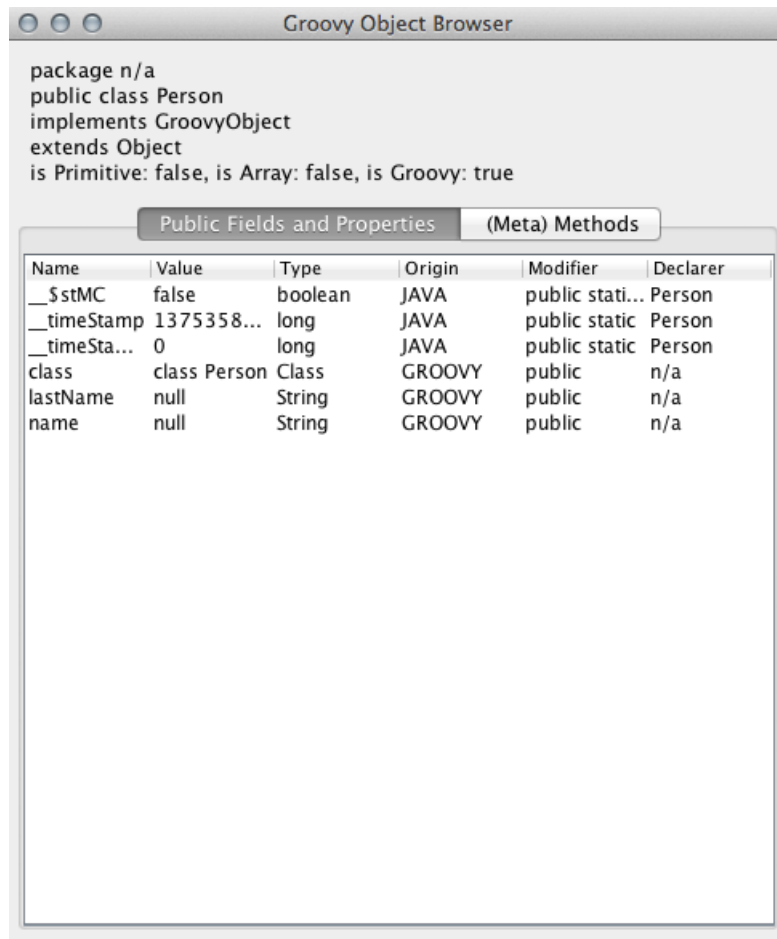
The "GroovyConsole" comes with a handy "Object Browser" that graphically shows the fields and methods available on a given class.

Press **Ctrl+I** (**CMD+I** on a **Mac**) to launch the browser and display the last evaluated expression. The following simple class would appear like in the image below:

```
class Person {
 String name
 String lastName
}

p = new Person()
```





9366OS\_01\_41.png

By pressing **Ctrl+J** (**Cmd+J** on a **Mac**) we can display another view of the Object Browser, this time focused on the variables visible from within the script.

## Configuring Groovy in Eclipse

**Eclipse** is a popular **IDE** that is taking a big share of the **IDE** market for **JVM**-based technologies as well as other languages and platforms. In fact **Eclipse** is a huge ecosystem devoted to building tooling for different areas: dynamic languages, reporting, testing, modeling, analysis etc.

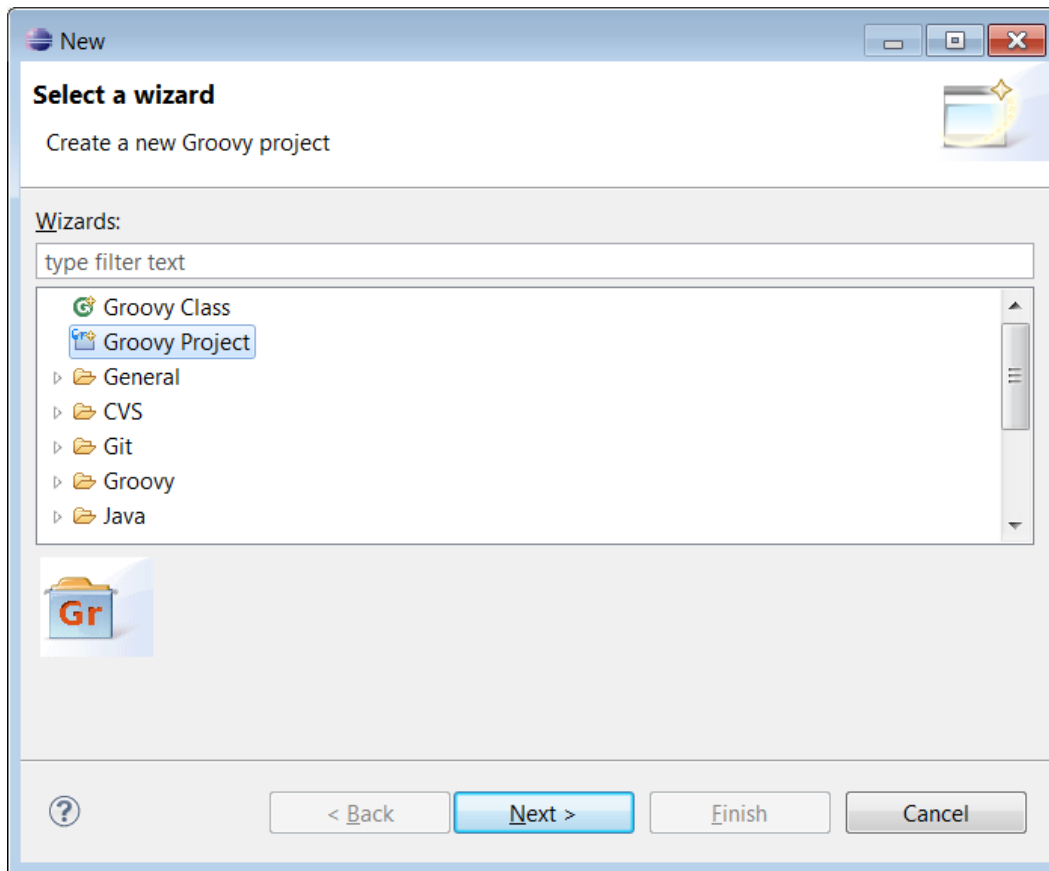
**Eclipse** is distributed in various combinations featuring different areas. A special distribution exists that comes prepackaged with **Eclipse** plugin for **Groovy** (code named **GRECLIPSE**) as well as various other useful plugins. It can be downloaded from <http://grails.org/products/ggts>.

However, in this recipe we will describe the installation instructions for the **Eclipse** plugin for **Groovy** assuming that it was not part of the distribution that you already have. These steps have been tested against **Eclipse Indigo 3.7**, but are very similar when using previous or later versions of **Eclipse**.

## How to do it...

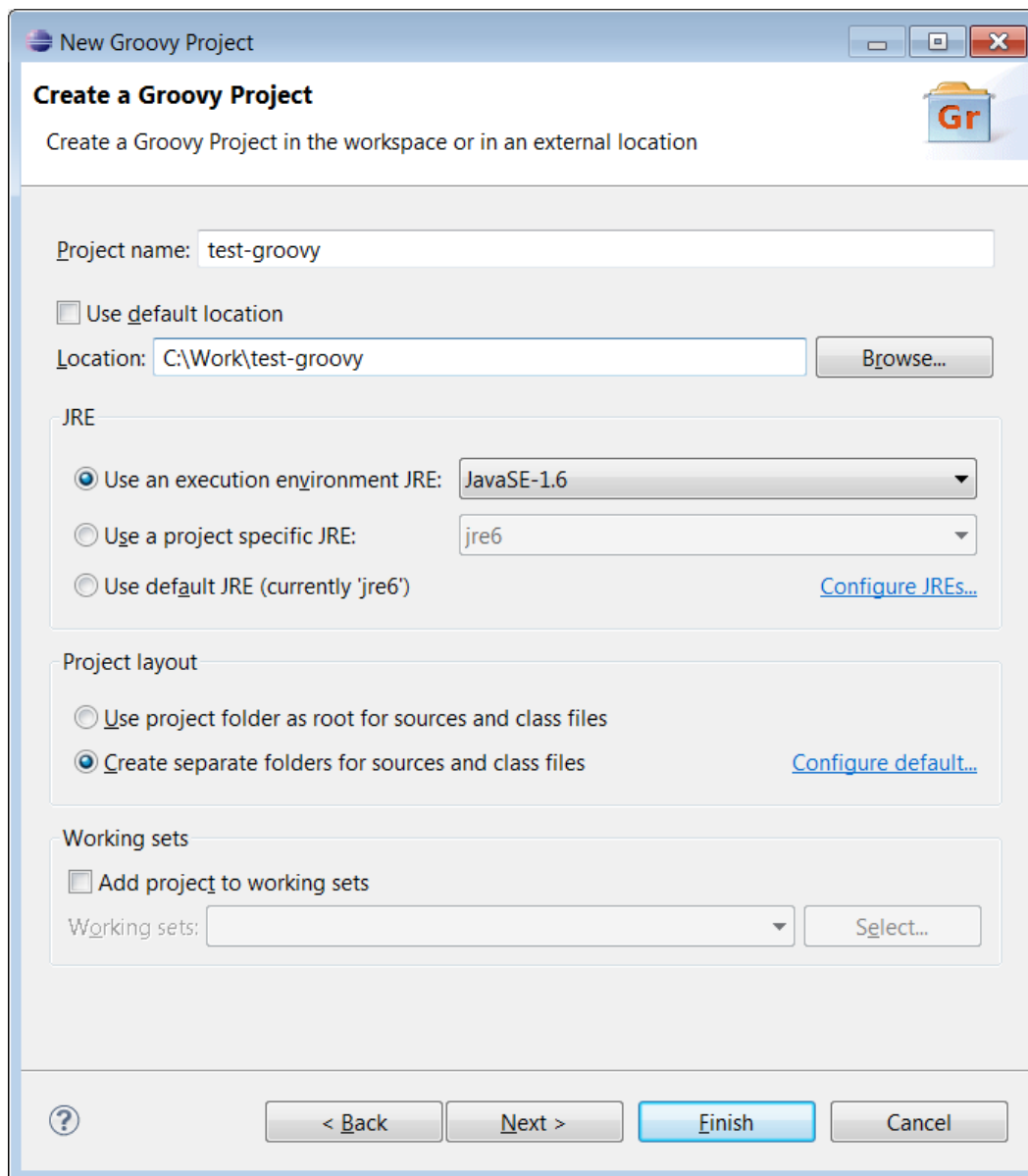
The plugin installation in **Eclipse** usually starts with opening "Install New Software..." menu item under "Help" menu.

1. In "Work with:" text box you need to type a plugin update site **URL**, which is <http://dist.springsource.org/release/GRECLIPSE/e3.7/> for the given version of **Eclipse**.
2. After you hit **Enter**, **Eclipse** will load the update site contents into the tree view.
3. You need to select the "Groovy-Eclipse" component and click "Next >" button.
4. Then you will have to follow the standard **Eclipse** installation procedure by accepting the software license and restarting **Eclipse** workspace after the download and installation process is complete.
5. When your **Eclipse** instance starts again, it should be possible to create **Groovy** projects as well as edit, compile and execute **Groovy** source files.
6. To get started, let's create a new project first. Go to "New" menu and select "Other..." menu item.
7. In "New" wizard window you should see "Groovy Project" item, which you should select and click "Next>" button:



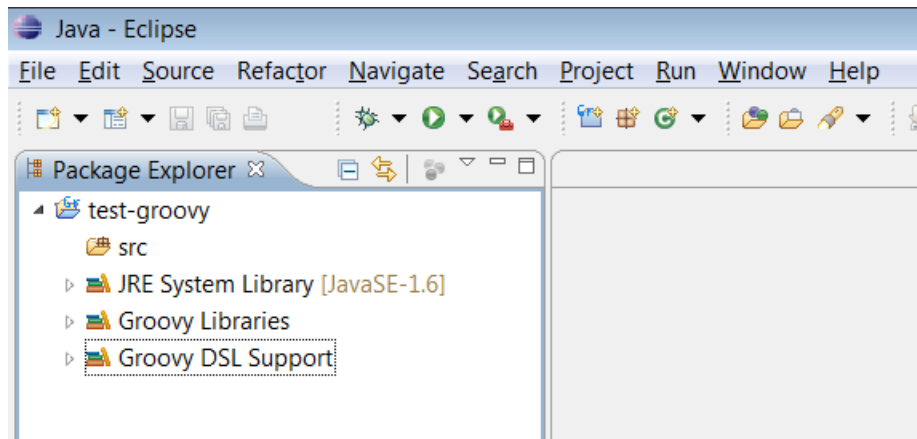
9366OS\_01\_03.png

8. The next dialog shows the new project's options, which are very similar to the one for **Java** projects:



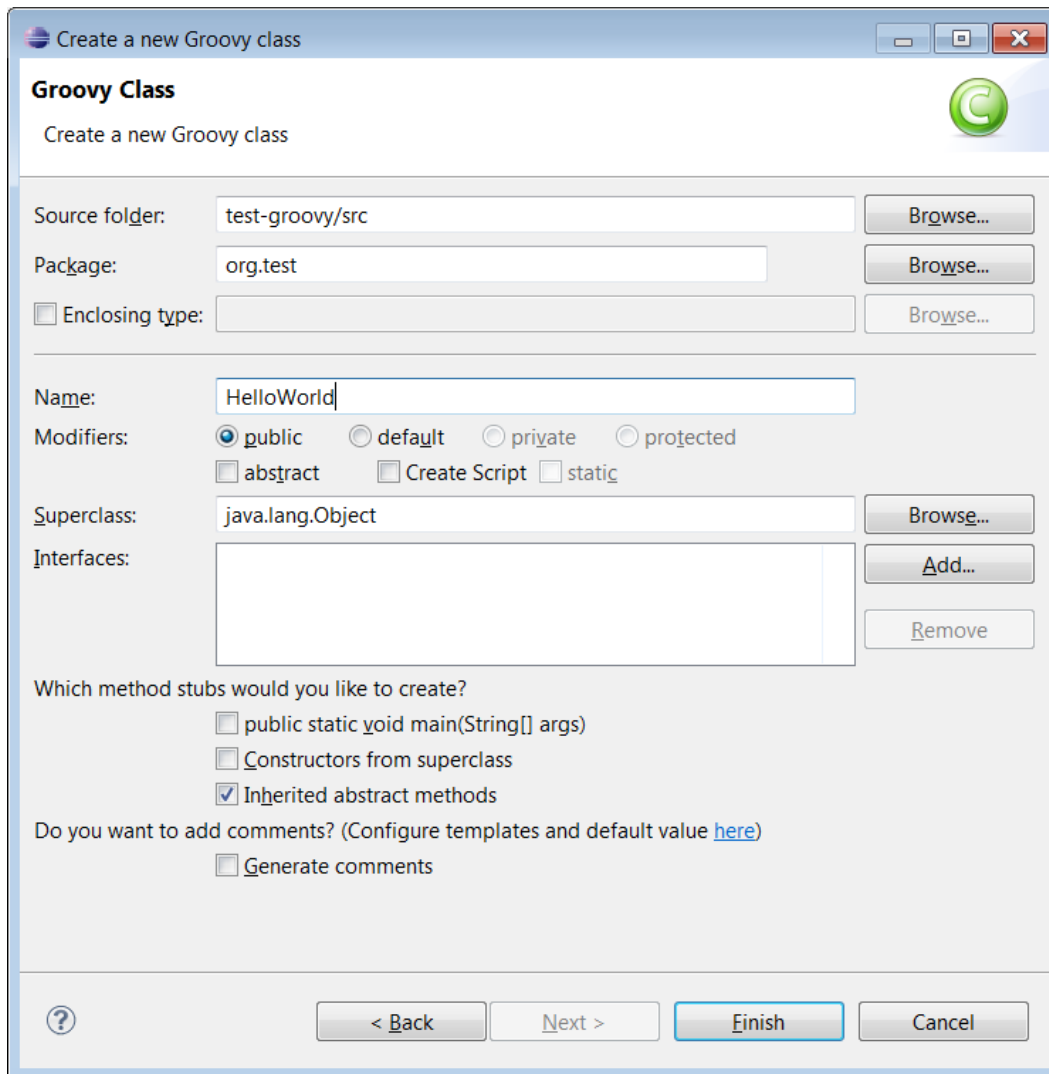
9366OS\_01\_04.png

You need to choose a project name (e.g. **test-groovy**), project location (e.g. **C:\work\test-groovy**), target **JRE** and click on the "Finish" button for the project to be created in the current work space:



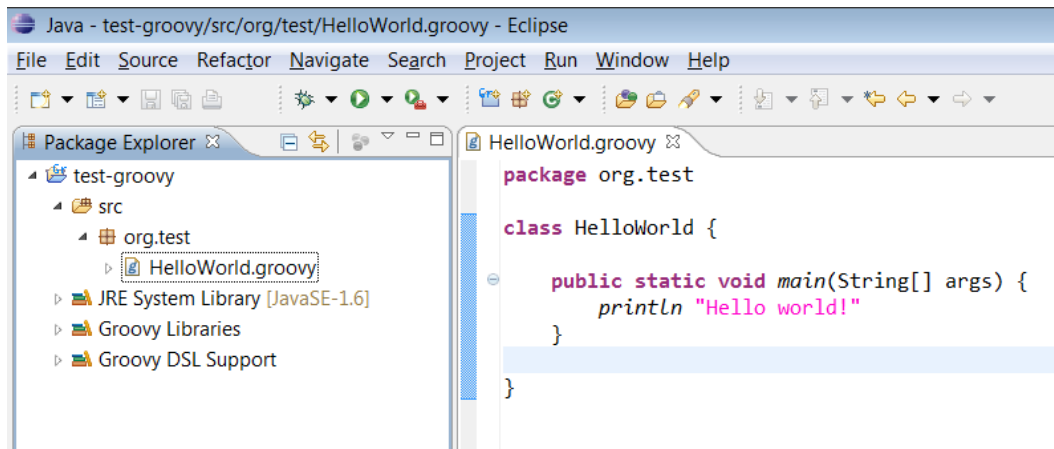
9366OS\_01\_05.png

9. Now it's time to add some code. You can do it through the same "New" menu and add new "Groovy Class" to the recently created project.
10. The class creation dialog has many similarities with the one for **Java** classes:



9366OS\_01\_06.png

11. **Groovy** shares almost 100% of **Java** syntax, therefore you can start creating **Groovy** classes in the same way you would do it in **Java**. And you will also get all the syntax highlighting and other **IDE** goodies when editing **Groovy** source files:

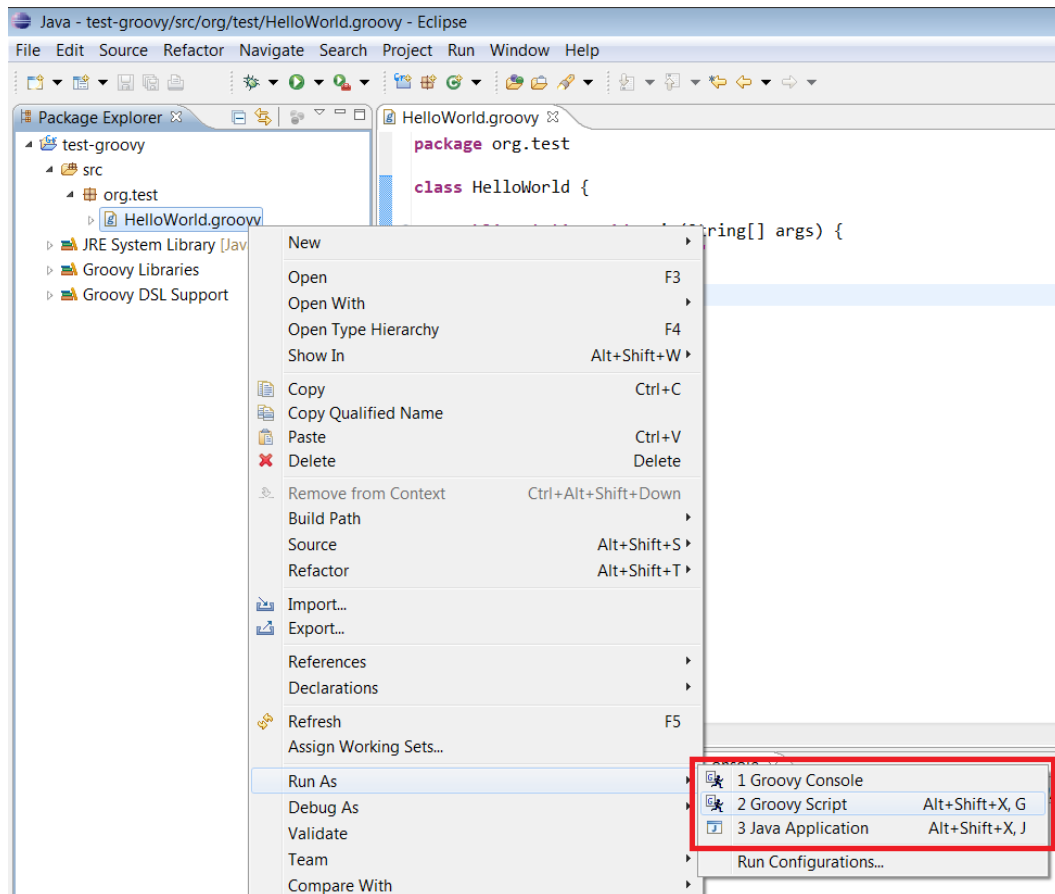


9366OS\_01\_07.png

## How it works...

The code shown in the previous section is actually **Groovy** code (even though it looks very much like **Java**) since we omitted semicolons and used the `println` method. The reasons for that method to be available were described in the *"Executing Groovy code from the command-line"* recipe.

With the **Eclipse** plugin for **Groovy** you also get the possibility to execute **Groovy** code right from your **IDE**. For that you need to use "Run as..." menu item from context menu:



9366OS\_01\_08.png

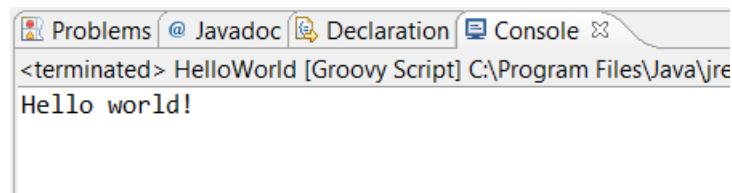
"Groovy Console" executor will send your code to `groovyConsole` instance, which is described in more details in *"Starting `groovyConsole` to execute Groovy snippets"* recipe.

The "{Groovy} Script" executor will run your code in the same way as `groovy HelloWorld.groovy` command would do it.

The "Java Application" executor (similarly to how it works for `Java` classes) will try to invoke the `main` method of your class.

By running the example code with either "{Groovy} Script" executor or "Java Application" executor, you should get the following output in Eclipse's "Console" view:



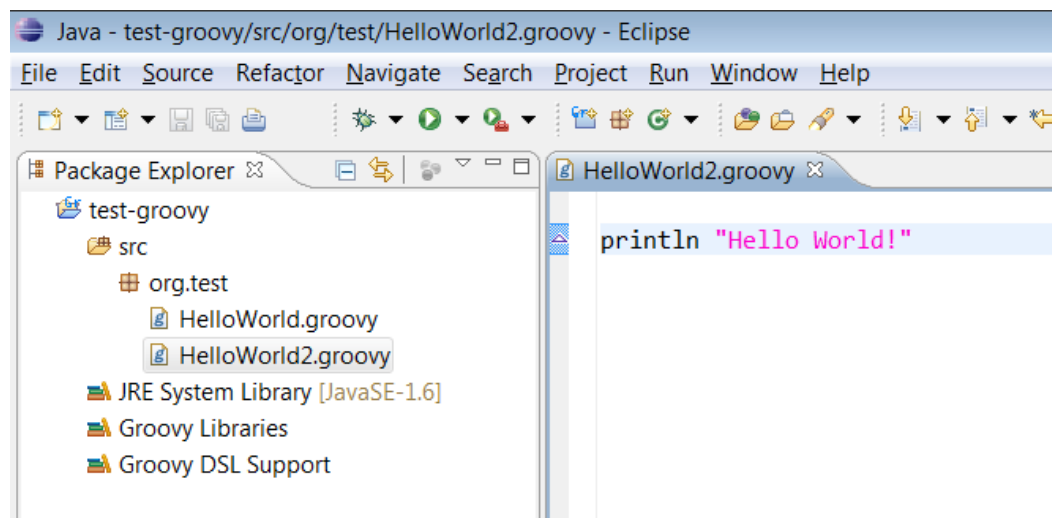


9366OS\_01\_09.png

## There's more...

**Groovy** can be used to write standard classes. But as we have seen in the previous recipes **Groovy** is also a scripting language, and the **Eclipse IDE** plugin supports this feature as well.

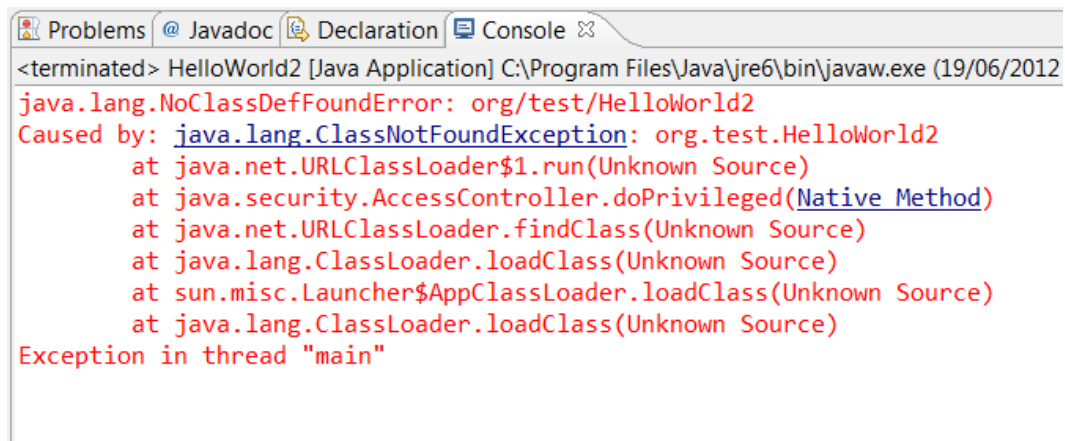
To define a **Groovy** script in your project, simply create a plain empty text file with the **\*.groovy** extension anywhere in your project. The new file is a **Groovy** script, so you can start by typing:



9366OS\_01\_10.png

When you run your script (**HelloWorld2.groovy**) with the "**{Groovy} Script**" executor you will get the same output as the previously mentioned **Groovy** source file (**HelloWorld.groovy**).

Conversely, if you run your **Groovy** script (**HelloWorld2.groovy**) using the "**Java Application**" executor, then it will complain about a missing class:

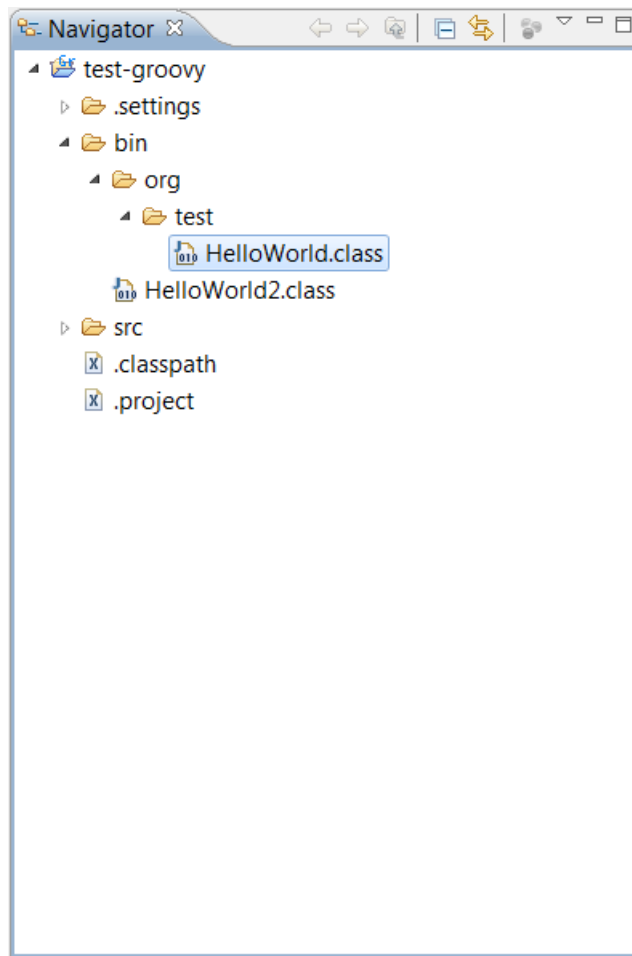


The screenshot shows the Eclipse IDE's Console window. The title bar includes tabs for Problems, Javadoc, Declaration, and Console. The console output shows a terminated Java application named 'HelloWorld2' running on 'C:\Program Files\Java\jre6\bin\javaw.exe' on '19/06/2012'. The error is a 'java.lang.NoClassDefFoundError: org/test/HelloWorld2'. The stack trace indicates the error was caused by a 'java.lang.ClassNotFoundException: org.test.HelloWorld2' and lists several frames from the Java runtime, including 'java.net.URLClassLoader\$1.run', 'java.security.AccessController.doPrivileged', 'java.net.URLClassLoader.findClass', and 'java.lang.ClassLoader.loadClass'. The final line of the stack trace is 'Exception in thread "main"'. The text is color-coded: red for error messages and blue for class names and method names.

```
<terminated> HelloWorld2 [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (19/06/2012)
java.lang.NoClassDefFoundError: org/test/HelloWorld2
Caused by: java.lang.ClassNotFoundException: org.test.HelloWorld2
 at java.net.URLClassLoader$1.run(Unknown Source)
 at java.security.AccessController.doPrivileged(Native Method)
 at java.net.URLClassLoader.findClass(Unknown Source)
 at java.lang.ClassLoader.loadClass(Unknown Source)
 at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
 at java.lang.ClassLoader.loadClass(Unknown Source)
Exception in thread "main"
```

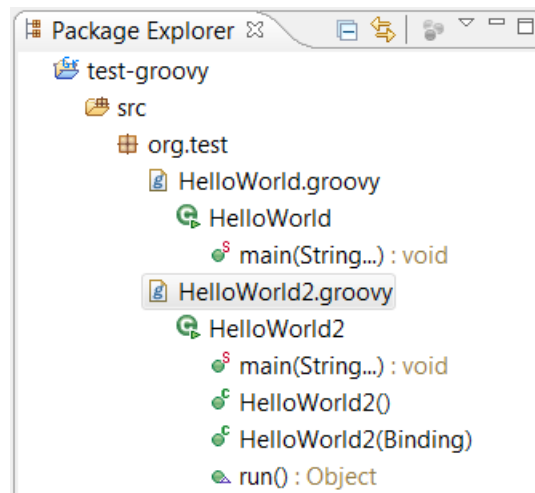
### 9366OS\_01\_11.png

The reason for the exception lays in how **Groovy** distinguishes between script files and classes. If you look at the way **Groovy** script and **Groovy** class are compiled in the **Eclipse "Navigator"** view, you can also notice that even though the script source file resides in the package folder **org.test** under **src** directory, that information is erased when the actual script is compiled into a class file:



9366OS\_01\_12.png

You can also spot the difference between scripts and classes by looking at the list of the class generated methods, by activating the "Package Explorer" view:



9366OS\_01\_13.png

As you can notice the **Groovy** script class has additional constructors as well as the **main** and **run** methods. The **Groovy** class has only the methods that were defined by the class definition.

This distinction makes **Groovy** a very powerful addition to **Java** tool set. **Groovy** allows you to use an alternative and more concise syntax to extend your existing **Java** applications as well as to write short and concise scripts that leverages the richness of **Java** platform.

## See also

Additional and updated information regarding the **Groovy** plugin for **Eclipse** can be found at <http://groovy.codehaus.org/Eclipse+Plugin>.

## Configuring Groovy in IntelliJ IDEA

Since version 8, **IntelliJ IDEA**, one of the most popular **Java IDE**, has native support for **Groovy**. There is no need to install any plugin to start coding in **Groovy** and benefit from a number of interesting features, such as code completion and refactoring. The latest iteration of **IntelliJ IDEA**, version **12**, has full support for **Groovy 2.0**.

In this recipe, we are going to show how to set up a **Groovy** project in **IDEA** and showcase some of the most interesting qualities of the integration.

## Getting ready...

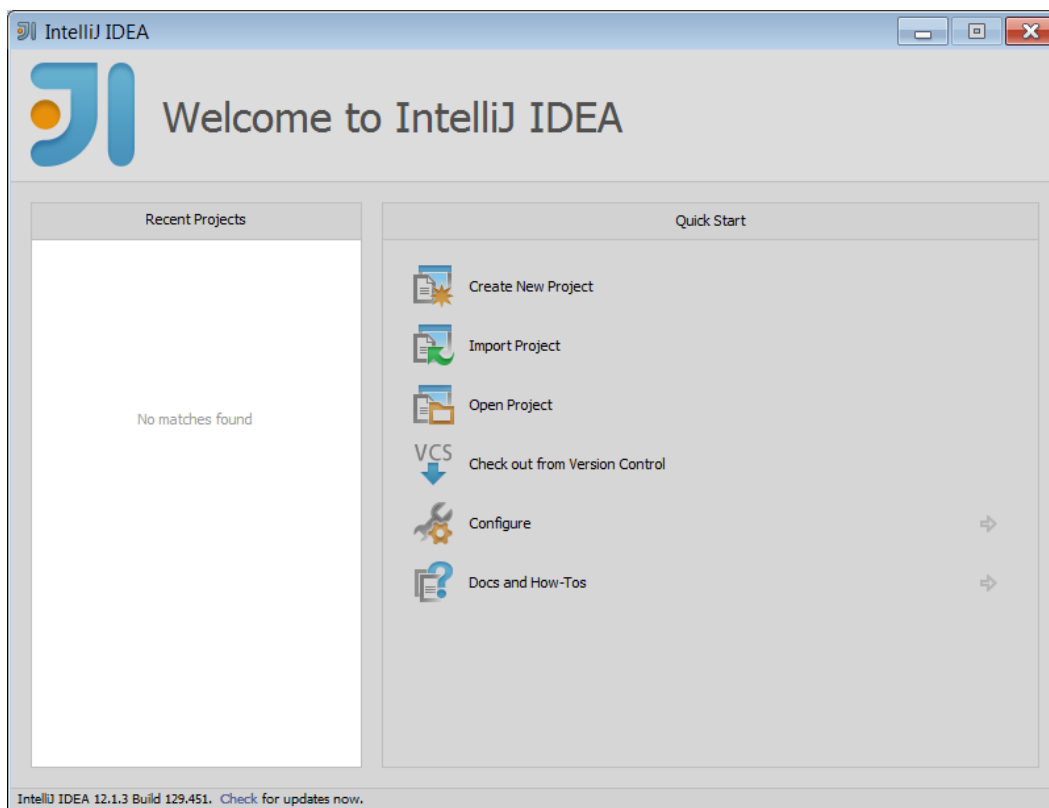
To get started with this recipe you need version **12** of **IntelliJ IDEA**. The **IDE** comes in two versions, "Community Edition" and "Ultimate". **JetBrains**, the company behind **IDEA**, offers the "Community Edition" for free, while it charges for the "Ultimate" version. The good news is that **Groovy** support is available in the free version of the **IDE**, so you can start using it straight away. Download **IntelliJ IDEA** from here: <http://www.jetbrains.com>.

You also need to install **Java** and a **Groovy** distribution. Refer to the installation recipes from this chapter.

## How to do it...

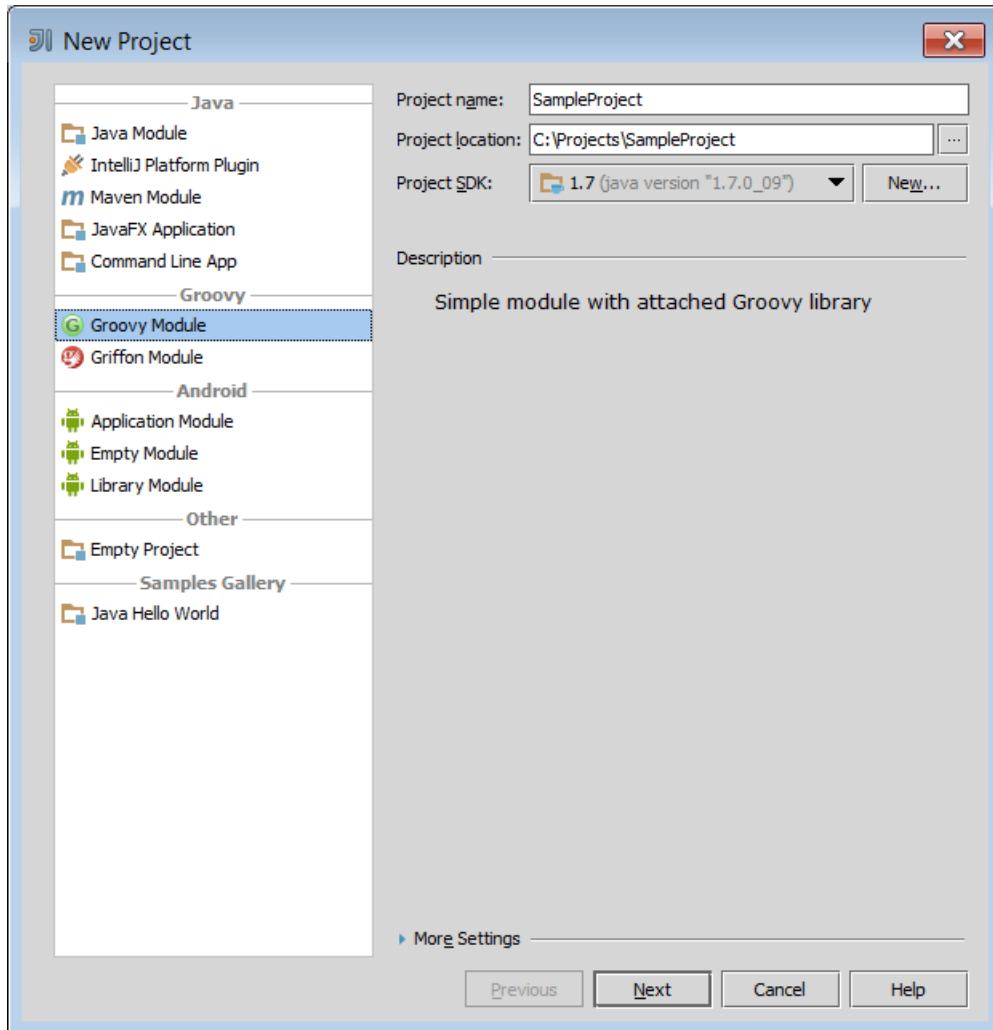
Let's start with project creation.

1. Click on the "Create New Project" link in the main **IntelliJ IDEA** start page:



## 9366OS\_01\_14.png

2. In the next wizard page, select "Groovy Module" and enter the name and the location of your project:

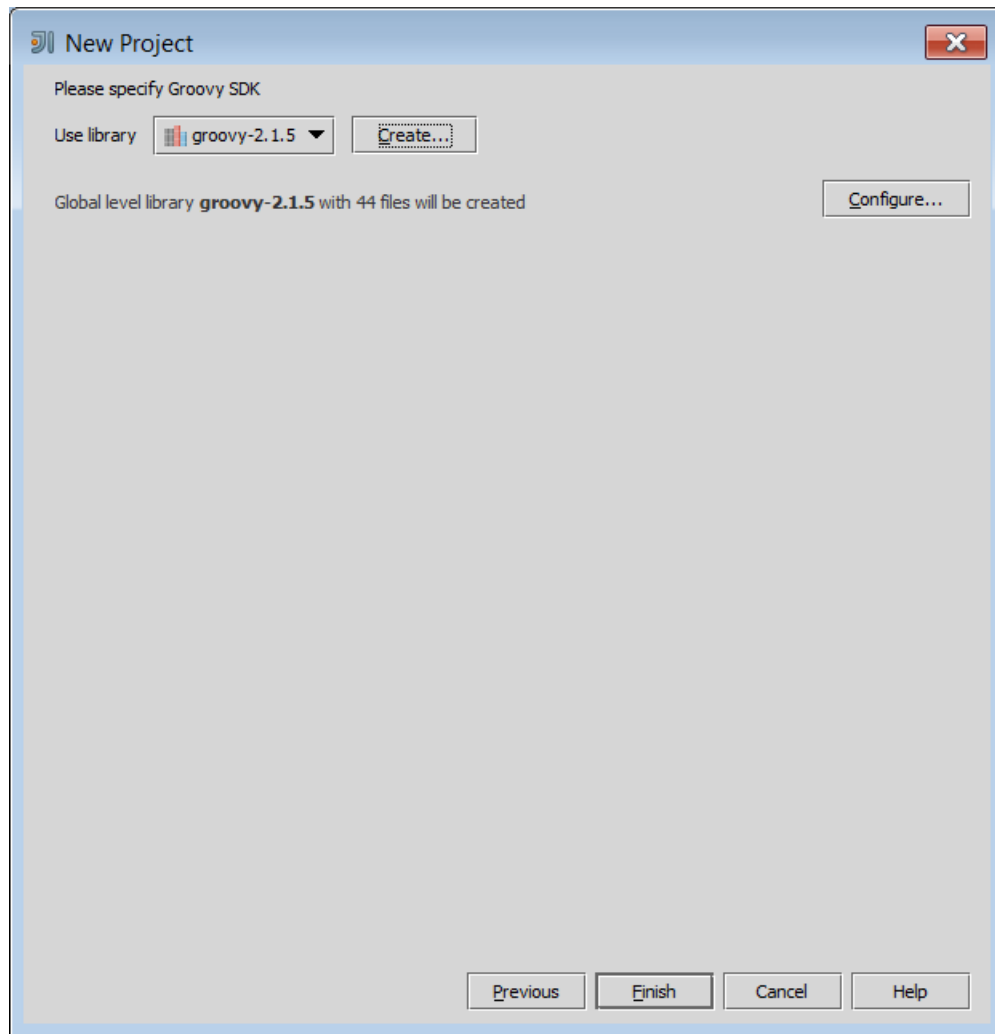


## 9366OS\_01\_15.png

3. The "Project SDK" field should be set to the version of **JDK** you want to use for this project. If "Project SDK" is empty and there is nothing in the list,

then press "New..." button, select "JDK" and locate the **JDK** installation directory in the opened folder selection dialog.

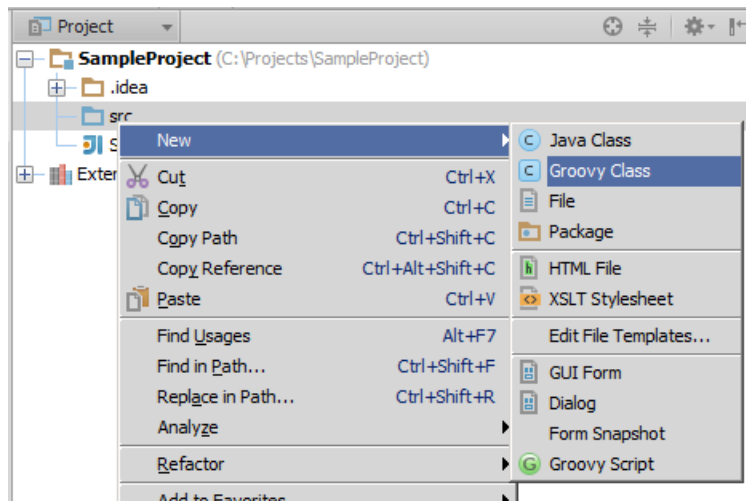
4. After **JDK** is selected click "Next" button to get to **Groovy SDK** selection page:



9366OS\_01\_16.png

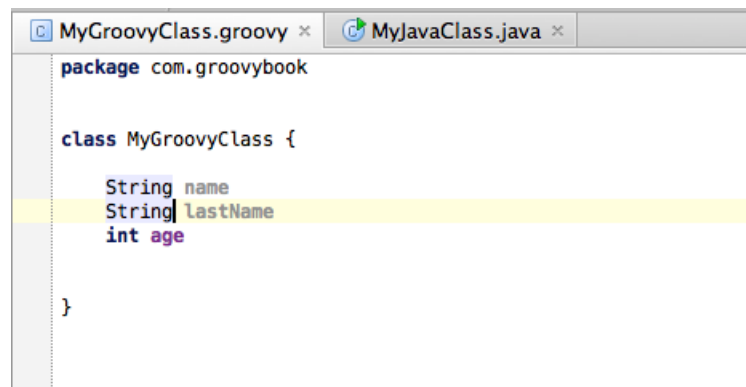
5. If it is a fresh installation, most likely the "Use library" drop down list will be empty. Click "Create" button and locate the **Groovy** installation directory. That's it, click on the "Finish" button.

6. Once the project is created, the **IDE** allows to create **Groovy** classes as well as **Groovy** scripts. Scripts can be used as *scrapbooks* for testing code snippets and can be later integrated into a class.



9366OS\_01\_17.png

7. You can also create **Java** classes that call methods on **Groovy** classes. Let's create a simple **Groovy** class, named **MyGroovyClass**:

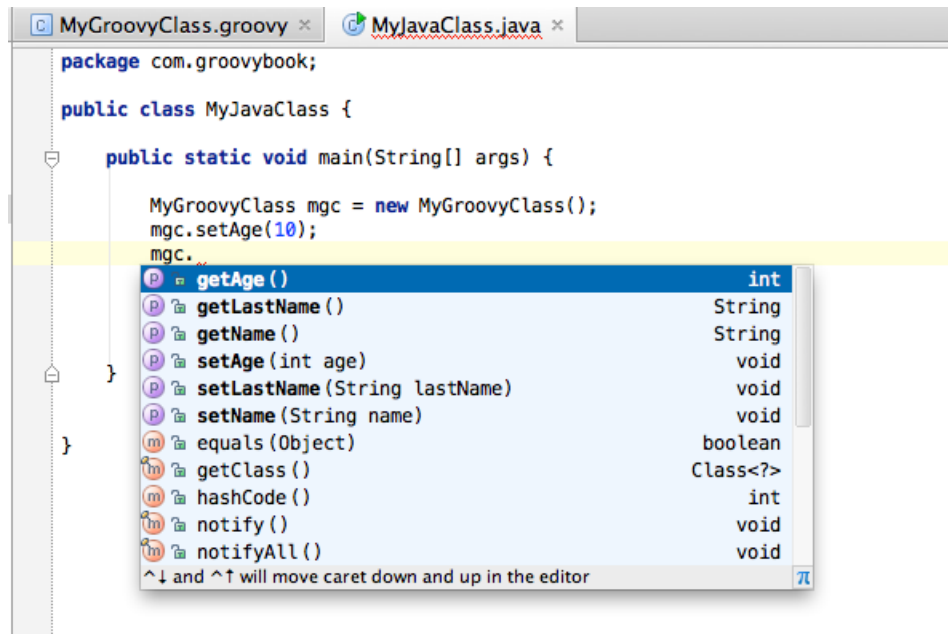


9366OS\_01\_18.png

8. The class defines three attributes and it uses the concise **Groovy** approach, no need to have getters and setters to access the variables.



9. Now let's create a **Java** class, `MyJavaClass`:



9366OS\_01\_19.png

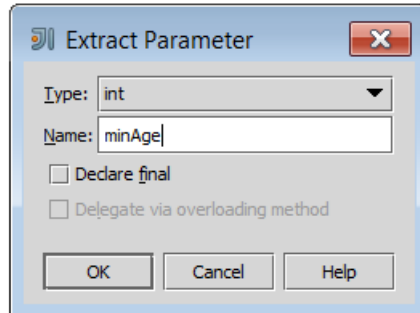
As you can see from the image above, the code autocompletion works perfectly and it's even able to propose synthetic methods that are generated by the **Groovy** compiler - such as getters and setters.

For a broader look at the integration between **Java** and **Groovy**, take a look at the *"Using Java classes from Groovy"* recipe.

## There's more...

Several refactoring goodies are also available, including **"Extract Parameter"** that works also for closures (see *"Defining code as data in Groovy"* recipe):

```
def isMinimalAge = { int age ->
 return age < 25
}
```



9366OS\_01\_20.png

The result of the refactoring will be the following:

```
def isMinimalAge = { int age, int minAge ->
 return age < minAge
}
```

9366OS\_01\_21.png