Advanced Java

Table of Contents

JShell . . . 2
- Introducing JShell . . . 2
- Reasons for JShell . . . 2
- JShell Prerequisites . . . 2
- Starting JShell . . . 2
- JShell in verbose mode . . . 3
- Trying out JShell . . . 3
- Replacing Variables in JShell . . . 3
- `final` is ignored . . . 3
- Scratch Variable . . . 4
- Creating methods . . . 4
- Changing method definitions . . . 5
- Creating a `class` . . . 5
- Making a mistake . . . 5
- Analyzing an `Exception` . . . 6
- Running an Editor . . . 7
- Running your own editor . . . 7
- Resetting your editor . . . 7
- Typical Commands . . . 7
- Viewing Commands . . . 7
- Tab Completion . . . 8
- Snippet Transformation . . . 8
- Input Line Navigation . . . 10
- Setting the Classpath . . . 11
- Scripting . . . 11
- Loading Scripts . . . 13
- Exiting JShell . . . 13
- Where to find more information . . . 13
- Lab: JShell . . . 14

Thank You . . . 15

JShell

Introducing JShell

- Java Shell tool (JShell) is an interactive tool for learning the Java programming language and prototyping Java code
- Read-Evaluate-Print Loop (REPL)
- Evaluates Expressions as they are entered
- JShell accepts Java
  - statements
  - variables
  - methods
  - `class` definitions
  - `import` definitions
  - expressions

Reasons for JShell

- Trial of code before implementation
- Establish and communicate ideas with code
- Bring production code and dissect problems
- Take ideas or fixes back to the IDE

JShell Prerequisites

- JDK 9 or higher

Starting JShell

```
% jshell
```

```
|  Welcome to JShell -- Version 10
|  For an introduction type: /help intro

jshell>
```

JShell in verbose mode

- `-v` will enter you in verbose mode
- Full description of actions performed within JShell

```
% jshell -v
```

Trying out JShell

- Notice there are no semicolons in using JShell when it deals with *some* assignments
- Later versions of Java can make using of `var` same in JShell

```
jshell> var list = List.of(3,4,5,6)
list ==> [3, 4, 5, 6]

jshell> var mapped = list.stream().map(x -> x + 1).collect(Collectors.toList())
mapped ==> [4, 5, 6, 7]

jshell>
```

Replacing Variables in JShell

In this mode you have the availability to reassign a variable in JShell

```
jshell> var x = List.of(1,2,3,4)
x ==> [1, 2, 3, 4]
|  created variable x : List<Integer>

jshell> var x = 30
x ==> 30
|  replaced variable x : int
|    update overwrote variable x : List<Integer>
```

`final` is ignored

- The keyword `final` is ignored as a top level assignment within JShell
- If the verbose is turned on in JShell using `-v` you can review the message

```
jshell> final var d = 30
|  Warning:
|  Modifier 'final'  not permitted in top-level declarations, ignored
|  final var d = 30;
|  ^---^
d ==> 30
|  created variable d : int
```

Scratch Variable

- If you do not create a variable name, one will be created for you
- This is called a *scratch variable*
- It is assigned with a variable `$n` where n is a monotonically increasing integer

```
jshell> 2 + 2
$3 ==> 4
|  created scratch variable $3 : int
```

It can then be subsequently called by that variable

```
jshell> $3 + 2
$4 ==> 6
|  created scratch variable $4 : int
```

Creating methods

- Methods can also be created *without* a surrounding `class`
- They can then be invoked afterwards by name with a surrounding `class`

```
jshell> public String times(int n, String s) {
   ...>    StringBuilder sb = new StringBuilder();
   ...>    for (int i = 0; i < n; i++) {
   ...>       sb.append(s);
   ...>    }
   ...>    return sb.toString();
   ...> }
```

Invocation can then be invoked using standard Java:

```
jshell> times(3, "Foo")
$6 ==> "FooFooFoo"
|  created scratch variable $6 : String
```

Changing method definitions

- If you want to rewrite a method, you can do so by just creating a different implementation
- If you have `-v` verbose mode on, this will show that you are replacing the definition

```
jshell> public String times(int n, String s) {
   ...>   return IntStream.range(0, n).boxed()
   ...>           .map(x -> s).collect(Collectors.joining());
   ...> }
|  modified method times(int,String)
|    update overwrote method times(int,String)
```

Creating a `class`

- Much like a method, a `class` can be created in JShell
- Merely type in the declaration and use the class at will

```
jshell> public class Country {
   ...>    private String name;
   ...>    private String capital;
   ...>    public Country (String name, String capital) {
   ...>      this.name = name;
   ...>      this.capital = capital;
   ...>    }
   ...>    public String toString() {
   ...>      return "Country {" + name + ", " + capital + "}";
   ...>    }
   ...> }
|  created class Country
```

Subsequently, you can then instantiate the class

```
jshell> var china = new Country("China", "Beijing");
china ==> Country {China, Beijing}
|  created variable china : Country

jshell> var poland = new Country("Poland", "Warsaw");
poland ==> Country {Poland, Warsaw}
|  created variable poland : Country
```

Making a mistake

If you make a mistake, you can hit <**UP-arrow**> and edit the previous unrunnable code

```
jshell> public String times(int n, String s) {
   ...>    StringBuilder sb = new StringBuilder()
   ...>    for (int i = 0; i < n; i++) {
   ...>       sb.append(s);
   ...>    }
   ...>    return sb.toString()
   ...> }
|  Error:
|  ';' expected
|     StringBuilder sb = new StringBuilder()
|                                          ^
|  Error:
|  ';' expected
|     return sb.toString()
|
```

- Hitting the <**UP-arrow**> key, it gives you the ability to edit again
- Typing <**Return**> anywhere in the code edit section will execute the method

```
jshell> public String times(int n, String s) {
   ...>    StringBuilder sb = new StringBuilder();
   ...>    for (int i = 0; i < n; i++) {
   ...>       sb.append(s);
   ...>    }
   ...>    return sb.toString()
   ...> }
| created method times(int,String)
```

Analyzing an `Exception`

- You can trace the root of the `Exception` by:
  - Reading the stack trace
  - Tracing the stack trace

In an exception back-trace, a location within REPL entered code is displayed as the `#id/linenumber`, where snippet id is the number displayed in `/list` and line-number is the line-number within the snippet.

```
jshell> int divide(int a, int b) {
...> return a / b;
}

jshell> divide(5, 0);
| java.lang.ArithmeticException thrown: / by zero
| at divide (#1:2)
| at (#2:1)
```

Running an Editor

- You can edit any code using `/edit` command along with a declaration.
- Declarations can either be `class`, method, or variable

Edit `Country`, in this case it is a `class`

```
jshell> /edit Country
```

This will in turn open an editor so that you can edit fully with full cursor support

[jshell edit] | *jshell_edit.png*

Figure 1. Standard Editor in JShell

Running your own editor

```
jshell> /set editor vi
```

Resetting your editor

```
jshell> /set editor -default
```

Typical Commands

- `/vars` will show all variables bound
- `/methods` will show all methods bound
- `/types` will show all types, e.g. classes
- `/list` shows a list of entered "snippets"

Viewing Commands

- We can view the possible list of commands by typing `/` and then <**TAB**>

```
jshell> /
/!          /?          /drop       /edit
/env        /exit       /help       /history
/imports    /list       /methods    /open
/reload     /reset      /save       /set
/types      /vars

<press tab again to see synopsis>

jshell> /
```

Tab Completion

Press <**TAB**> after some code to see some auto-complete alternatives

```
jshell> System.out.
append(         checkError()    close()         equals(
flush()         format(         getClass()      hashCode()
notify()        notifyAll()     print(          printf(
println(        toString()      wait(           write(
```

Type some more of the signature and get documentation

```
jshell> System.out.format(
Signatures:
PrintStream PrintStream.format(String format, Object... args)
PrintStream PrintStream.format(Locale l, String format, Object... args)

<press tab again to see documentation>
```

Pressing <**TAB**> again will show the full documentation

Snippet Transformation

- When invoking the keys combination of <**SHIFT**>+<**TAB**>, you can then use the following to transform your line
  - **m** will create a method
  - **v** will create a variable
  - **i** will provide a selection of `import` for a `package` of a class

Variable Snippet Transformation

- Given that we have already typed the following
- And our cursor is at the end of the line

- We can type <**SHIFT**>+<**TAB**> and then **v** to create a variable
- The cursor will be in a position to create the variable

```
jshell> new BigInteger("302021")
```

- After <**SHIFT**>+<**TAB**> and then <**v**>
  - The cursor will be positioned before the `=`
  - You now have the opportunity to enter a variable name

```
jshell> BigInteger  = new BigInteger("302021")
```

If we named it `i`, this would result in…

```
jshell> BigInteger i = new BigInteger("302021")
```

Method Snippet Transformation

- After creating snippet and our cursor is at the end of the line
- We can type <**SHIFT+TAB**> and then <**m**> to create a method
- The cursor will be in a position to create the method
- This will only work if all variables can be resolve

Establishing a variable

```
jshell> BigInteger i = new BigInteger("302021")
i ==> 302021
|  created variable i : BigInteger
```

We reference `i` and we realize we want as a method

```
jshell> i.add(new BigInteger("4")
```

- The following will give you the opportunity to name the method
- It will place the cursor before the `()`

After <**SHIFT+TAB**> *and* <**m**>

```
jshell> BigInteger () { return i.add(new BigInteger("4")); }
```

Import Snippet Transformation

- <**SHIFT+TAB**> and <**i**> will provide choices for `import` if it is not in the *java.base* module

- Type the `class` you need and at the end **<SHIFT+TAB>** and **<i>**.
- Select the package you wish to import

<SHIFT+TAB> *and* **<i>** *after typing* `DriverManager`

```
jshell> DriverManager
0: Do nothing
1: import: java.sql.DriverManager
Choice:
```

Input Line Navigation

Editing is supported for editing:

- The current line
- Accessing the history through previous sessions of JShell.
- **<CTRL>** and **<META>** key are used in key combinations.

Meta key is a key like the Windows key or Apple key, if not available, then use the **<ALT>** key

Editing Navigation

- When navigating forwards and backwards within a line:
  - Use the **<RIGHT-Arrow>** and **<LEFT-Arrow>**
  - or, **<CTRL+B>** or **<CTRL+F>**
- When bringing up previous snippets and commands use **<UP-Arrow>**
- If the previous commands or snippets contains multiple lines you can edit that line with **<UP-Arrow>** or **<DOWN-Arrow>**

Additional Keys for Editing Navigation

Keys	Action
<Return>	Enters the current line
<LEFT-arrow>	Moves backward one character
<RIGHT-arrow>	Moves forward one character
<UP-arrow>	Moves up one line, backward through history
<DOWN-arrow>	Moves down one line, forward through history
<CTRL+A>	Moves to the beginning of the line
<CTRL+E>	Moves to the end of the line
<META+B>	Moves backward one word

Keys	Action
<META+F>	Moves forward one word

Setting the Classpath

- You can use external code that is accessible through the class path in your JShell session.
- Use `--class-path` to load external directories and jar files

```
% jshell --class-path myOwnClassPath
```

While in an already loaded session you can use `/env --class-path`

```
jshell> /env --class-path myOwnClassPath
| Setting new options and restoring state.
```

To view the current classpath:

```
jshell> /env
|     --class-path guava-27.0.1-jre.jar
```

Scripting

- A JShell script is a sequence of snippets and JShell commands in a file, one snippet or command per line.
- Scripts can be a local file, or one of the following predefined scripts:

Script Name	Script Contents
DEFAULT	Includes commonly needed import declarations. This script is used if no other startup script is provided.
PRINTING	Defines JShell methods that redirect to the `print`, `println`, and `printf` methods in `PrintStream`.
JAVASE	Imports the core Java SE API defined by the `java.se` module, which causes a noticeable delay in starting JShell due to the number of packages.

Startup Scripts

- The default startup script has common imports, e.g. `java.lang`
- You can create custom imports as needed
- Startup scripts are loaded everytime:

- JShell is started
- When `/reset`, `/reload`, or `/env` commands are invoked
- The `DEFAULT` script is used by default

Setting up the Startup Script

- You can set the startup script using the following script
- This will only be active in the current session

```
jshell> /set start mystartup.jsh
jshell> /reset
| Resetting state.
jshell
```

You can retain the setup using `-retain` for subsequent sessions

```
jshell> /set start -retain DEFAULT PRINTING
```

Showing what has been retained

Use `/set start` with no arguments to show startup scripts

```
jshell> /set start
|  /set start -retain DEFAULT PRINTING
|  ---- DEFAULT ----
|  import java.io.<strong>;
|  import java.math.</strong>;
|  import java.net.<strong>;
|  import java.nio.file.</strong>;
```

Start Multiple Scripts from Command Line

- If starting from your shell command line, you can use the `--startup` flag
- This will establish startup scripts when first running `jshell`

```
% jshell --startup DEFAULT --startup PRINTING
```

Creating Scripts

- Scripts can be created in an editor or generated in JShell
- To save the JShell session, use either of the following:

```
jshell> /save mysnippets.jsh
```

Saves the history of all of the snippets and commands, both valid and invalid

```
jshell> /save -history myhistory.jsh
```

Saves the contents of the current startup script setting to *mystartup.jsh*.

```
jshell> /save -start mystartup.jsh
```

Loading Scripts

Load a script when first starting JShell

```
% jshell mysnippets.jsh
```

Within JShell a script can be loaded with `/open`

```
jshell> /open mysnippets.jsh
```

Exiting JShell

```
jshell> /exit
```

Where to find more information

- We covered some of the major functionality of JShell
- More fine-grained information can be found in the Official JShell Documentation

Lab: JShell

Step 1: Launch JShell in verbose mode

Step 2: Create a method or `class` that given a `List<String>` of people's name it would return a random winner. Use `java.util.Random` for the randomization

Step 3: Create variables `thirdPlace`, `secondPlace` and `firstPlace` that will return the winners from the `List` of `String`

Step 4: Exit the JShell

Thank You

- Email: dhinojosa@evolutionnext.com
- Github: https://www.github.com/dhinojosa
- Twitter: http://twitter.com/dhinojosa
- Linked In: http://www.linkedin.com/in/dhevolutionnext