1.  [Java 9 REPL (JShell)](https://javanotesbybalakrishnareddy.blogspot.in/p/jshell.html)  
2.   Java 9 Module System

3.   Process API Improvements

4.   Try With Resources Improvement

5.   Factory Methods for Immutable List, Set, Map and Map.Entry

6.   CompletableFuture API Improvements

7.   Reactive Streams

8.   Diamond Operator for Anonymous Inner Class

9.   Optional Class Improvements

10. Stream API Improvements

11. Enhanced @Deprecated annotation

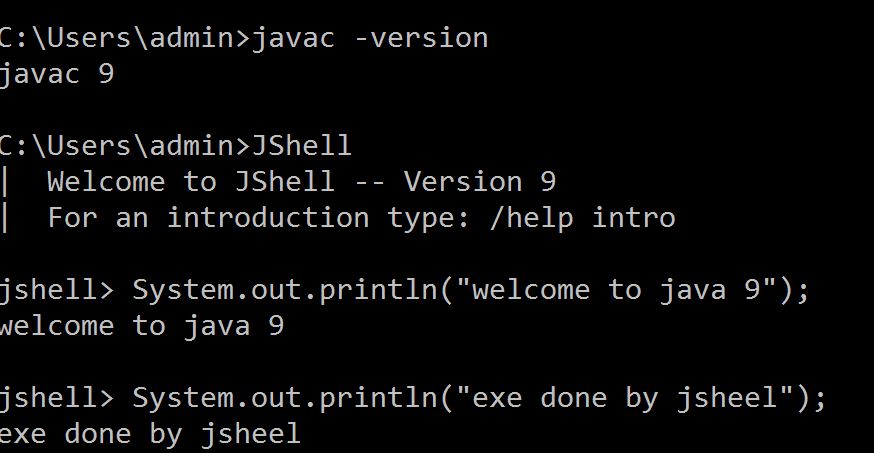
12. [Private methods inInterfaces](https://javanotesbybalakrishnareddy.blogspot.in/p/private-methods-in-interfaces.html)

13. Debugging in Production

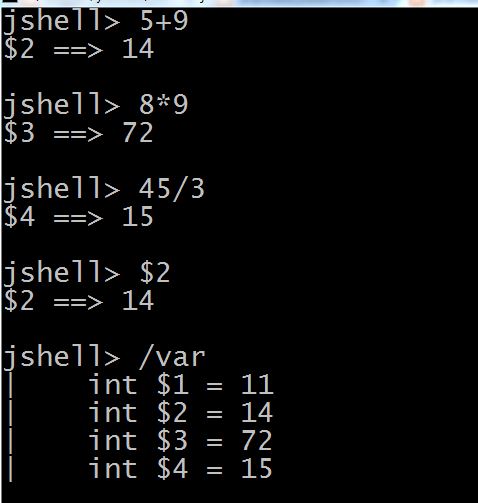
1.  [Java 9 REPL (JShell)](https://javanotesbybalakrishnareddy.blogspot.in/p/jshell.html):

* ***REPL* (Read-Evaluate-Print-Loop)**
* **REPL is an interactive shell into which we can enter commands and have these immediately executed and the results displayed**
* **quickly testing code without needing to write an entire test class to run.**
* **for statements interpretation**
* **Using these tools you can test code snippets rapidly without creating project**
* **using REPL each statement is executed instantly**

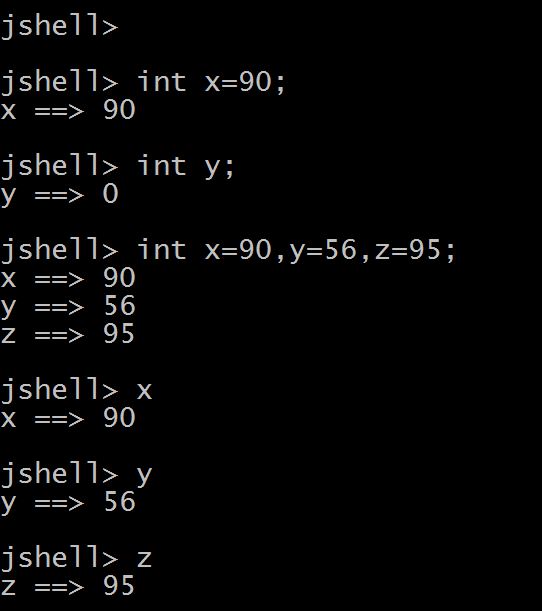
***How to start Jshell***



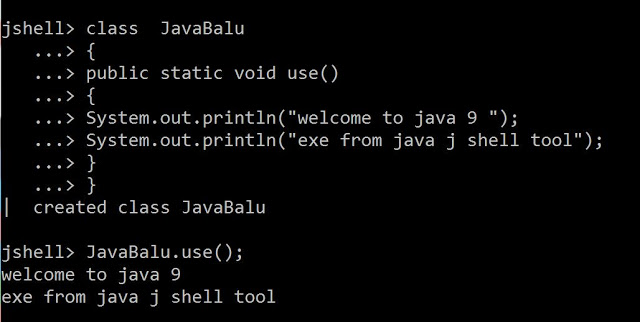
***How to work with Expressions***



***How to work with Variables***



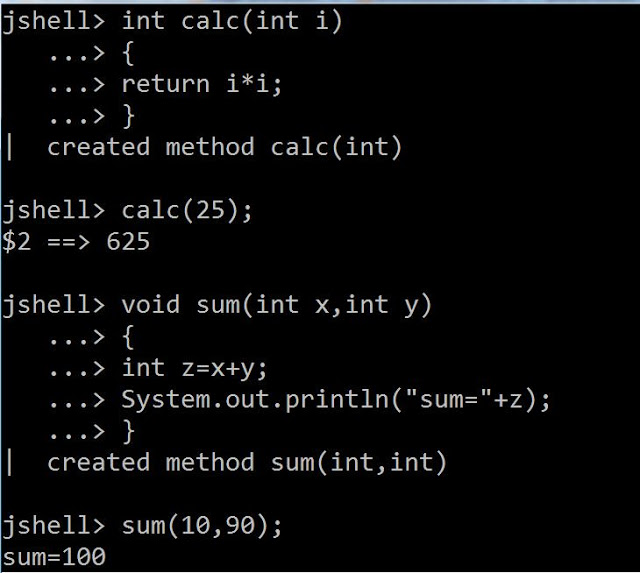
***How to Execute  Jshell class***

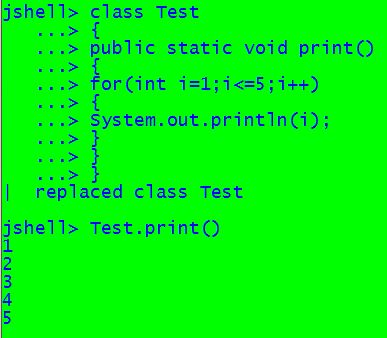


***Jshell Default import***

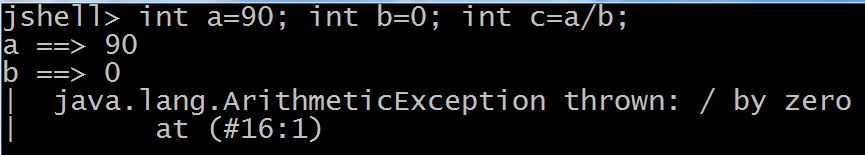


[***Working with Function***](https://4.bp.blogspot.com/-wM_99zGZhJ4/Wb6EqQN7nAI/AAAAAAAAApc/x8opvQYIVrQnu5QfQ1p2Clw9BG5gZgiOACLcBGAs/s1600/jsheel2.JPG)

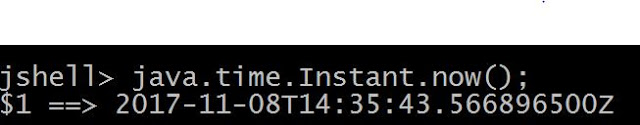
[](https://2.bp.blogspot.com/-JpFLzbjRfxc/Wb6bkT6pP9I/AAAAAAAAAqI/xEWuSR3t4icy92M12zsZojiyyKSgga_WwCLcBGAs/s1600/jsheel5.JPG)



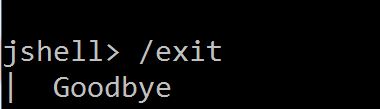
## *Exception Handling*



[***How to Know Time from  Jshell***](https://4.bp.blogspot.com/-wM_99zGZhJ4/Wb6EqQN7nAI/AAAAAAAAApc/x8opvQYIVrQnu5QfQ1p2Clw9BG5gZgiOACLcBGAs/s1600/jsheel2.JPG)

[](https://1.bp.blogspot.com/-dvDW7xtFkUw/WgMXPcwUmLI/AAAAAAAABRA/5ccduWWr3OMibnOAh4j14d4sRRLv5OQWwCLcBGAs/s1600/Capture.JPG)

[***How to Exit  Jshell***](https://4.bp.blogspot.com/-wM_99zGZhJ4/Wb6EqQN7nAI/AAAAAAAAApc/x8opvQYIVrQnu5QfQ1p2Clw9BG5gZgiOACLcBGAs/s1600/jsheel2.JPG)



Commands

Aparat from language syntax you can execute jshell commands. Some of the most useful ones (/help to list all of them) are:

listing variables

jshell> /vars

| int x **=** 0

| double j **=** 0.5

listing methods:

jshell> /methods

| printf **(**String,Object...**)**void

| helloJShell **()**void

The printf method is defined by default.

listing sources

jshell> /list

14 : helloJShell**()**;

15 : void helloJShell**()** **{** System.out.println**(**"wow, I replaced a method"**)**; **}**

16 : helloJShell**()**

editing sources in external editor

jshell> /edit helloJShell

Opens external editor, and replaces helloJShell method.

Example use cases

After 20 years of Java without REPL one might wonder what scenarios are suitable for JShell. Here are some examples.

Veryfing return type

Remember the time you learned that dividing two integers in Java does not result in floating number? For some time I was convinced that both numerator and denominator have to be floating for a result to be floating too. Let’s test that:

jshell> 1/2

$1 **==**> 0

| created scratch variable $1 : int

jshell> 1.0/2

$2 **==**> 0.5

| created scratch variable $2 : double

jshell> 1/2.0

$3 **==**> 0.5

| created scratch variable $3 : double

jshell> 1.0f/2

$4 **==**> 0.5

| created scratch variable $4 : float

jshell> 1/2.0f

$5 **==**> 0.5

| created scratch variable $5 : float

Turns out only one of them has to be floating.

Testing Java niuanses

Did you know that comparing autoboxed integers references which values are from range -128 to 127 (inclusive) returns true (they are cached)? You can verify that with shell in a matter of seconds:

jshell> Integer i1 **=** 127

i1 **==**> 127

jshell> Integer i2 **=** 127

i2 **==**> 127

jshell> i1 **==** i2

$35 **==**> true

jshell> Integer i2 **=** 128

i2 **==**> 128

jshell> Integer i1 **=** 128

i1 **==**> 128

jshell> i1 **==** i2

$38 **==**> false

Formatting

Sometimes the logs need to be verbose and properly formatted. This is tedious task and usually leads to few recompile cycles which significantly slows us down. Imagine you forgot what was the format sign responsible for integers. You can quickly verify that:

Let’s try %i (integer):

jshell> printf**(**"I got %i apple",1**)**

| java.util.UnknownFormatConversionException thrown: Conversion **=** 'i'

| at Formatter$FormatSpecifier.conversion **(**Formatter.java:2691**)**

| at Formatter$FormatSpecifier.<init> **(**Formatter.java:2717**)**

| at Formatter.parse **(**Formatter.java:2565**)**

| at Formatter.format **(**Formatter.java:2507**)**

| at PrintStream.format **(**PrintStream.java:977**)**

| at PrintStream.printf **(**PrintStream.java:873**)**

| at printf **(***#s8:1)*

| at **(***#51:1)*

Oops, maybe %d (decimal) :

jshell> printf**(**"I got %d apple",1**)**

I got 1 apple

## ****Exception Handling****

Anytime we run a command in JShell that ends up in an Exception reaching the top of the stack, this is automatically handled by displaying the stack trace, including all line numbers, similar to how Java [typically handles](https://stackify.com/specify-handle-exceptions-java/) exceptions.

This can be quite helpful for easily diagnosing problems:

==> String test(String input){

...> String upper = input.toUpperCase();

...> return upper.substring(10);

...> }

| created method test(String)

==> test("Hello");

| java.lang.StringIndexOutOfBoundsException thrown: String index out of range: -5

| at String.substring (String.java:1852)

| at test (#1:3)

| at (#2:1)

==>

This tells us an exception was thrown on line 3 of the test() method, which itself was command 1 of the script. We can then see what this was:

==> /list test

1 : String test(String input){

String upper = input.toUpperCase();

return upper.substring(10);

}

==>

Therefore, we can immediately see what the offending statement was. Unfortunately, the lack of line numbers means that this is easiest when we have short functions to work with, which is, of course, good practice anyway.

**Factory Methods for Immutable List, Set, Map and Map.Entry**

Oracle Corp has introduced some convenient factory methods to create Immutable List, Set, Map and Map.Entry objects. These utility methods are used to create empty or non-empty Collection objects.

In Java SE 8 and earlier versions, We can use Collections class utility methods like unmodifiableXXX to create Immutable Collection objects. For instance, if we want to create an Immutable List, then we can use Collections.unmodifiableList method.

However these Collections.unmodifiableXXX methods are very tedious and verbose approach. To overcome those shortcomings, Oracle corp has added couple of utility methods to List, Set and Map interfaces.

List and Set interfaces have “of()” methods to create an empty or no-empty Immutable List or Set objects as shown below:

**Empty List Example**

List immutableList = List.of();

**Non-Empty List Example**

List immutableList = List.of("one","two","three");

Map has two set of methods: of() methods and ofEntries() methods to create an Immutable Map object and an Immutable Map.Entry object respectively.

**Empty Map Example**

jshell> Map emptyImmutableMap = Map.of()

emptyImmutableMap ==> {}

**Non-Empty Map Example**

jshell> Map nonemptyImmutableMap = Map.of(1, "one", 2, "two", 3, "three")

nonemptyImmutableMap ==> {2=two, 3=three, 1=one}

### Java 9 Module System

One of the big changes or java 9 feature is the Module System. Oracle Corp is going to introduce the following features as part of **Jigsaw Project**.

* Modular JDK
* Modular Java Source Code
* Modular Run-time Images
* Encapsulate Java Internal APIs
* Java Platform Module System

Before Java SE 9 versions, we are using Monolithic Jars to develop Java-Based applications. This architecture has lot of limitations and drawbacks. To avoid all these shortcomings, Java SE 9 is coming with Module System.

JDK 9 is coming with 92 modules (may change in final release). We can use JDK Modules and also we can create our own modules as shown below:

**Simple Module Example**

module com.foo.bar { }

Here We are using ‘module’ to create a simple module. Each module has a name, related code and other resources.

**Steps to Develop a Java 9 Module**

We will follow these steps one by one to develop and test our “HelloWorld” Module.

* Create Module name folder, for example “com.hello”.
* Create Module Packages, for example “com.hello”.
* Create our Java component, for example “HelloWorld.java”.
* Create Module Descriptor, for example “module-info.java”.
* Define Module Description in Module Descriptor, for example “exports com.hello;” of “module-info.java” in “com.hello” module.
* Create Module Jars if required.
* Test our modules.

These steps are common for almost all modules development. Most of the commands used in this post are applicable to Windows OS. Few commands may differ for other OSs like Linux, Mac.

**Develop “HelloWorld” Module**

We first start with “HelloWorld” Module development. Please refer our Problem statement diagram for more details.

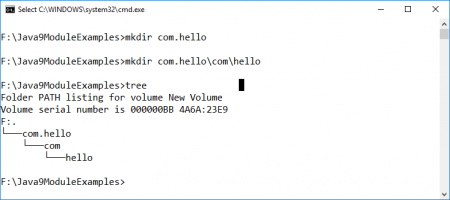
Create “HelloWorld” Module name folder: com.hello

mkdir com.hello

Create “HelloWorld” Module package name folder: com\hello

mkdir com.hello\com\hello

Please refer the following diagram for the above two steps:



Develop “HelloWorld.java” component under package name “com.hello\com\hello”.

**HelloWorld.java**

package com.hello;

public class HelloWorld {

public String sayHelloWorld() {

return "Hello World!";

}

}

Develop Module Descriptor at Module root folder “com.hello”.

**module-info.java**

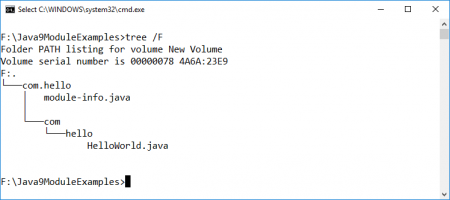
module com.hello {

exports com.hello;

}

If we observe this Module Descriptor, we can say that “com.hello” is exporting “com.hello” package to outside world so that our HelloWorldClient program can use it.

“com.hello” Module full tree structure as shown below:



**Develop “HelloWorldClient” Module**

Like “HelloWorld” Module, We need to follow the same steps to develop this module. Please refer our Problem statement diagram for more details.

Create “HelloWorldClient” Module name folder: com.hello.client

mkdir com.hello.client

Create “HelloWorldClient” Module package name folder: com\hello\client

mkdir com.hello\com\hello\client

Develop “HelloWorldClient.java” component under package name “com.hello\com\hello\client”.

**HelloWorldClient.java**

**Try With Resources Improvement**

We know, Java SE 7 has introduced a new exception handling construct: Try-With-Resources to manage resources automatically. The main goal of this new statement is “Automatic Better Resource Management”.

Java SE 9 is going to provide some improvements to this statement to avoid some more verbosity and improve some Readability.

**Java SE 7 example**

void testARM\_Before\_Java9() throws IOException{

BufferedReader reader1 = new BufferedReader(new FileReader("journaldev.txt"));

try (BufferedReader reader2 = reader1) {

System.out.println(reader2.readLine());

}

}

**Java 9 example**

void testARM\_Java9() throws IOException{

BufferedReader reader1 = new BufferedReader(new FileReader("journaldev.txt"));

try (reader1) {

System.out.println(reader1.readLine());

}

}