



---

# The Java Shell (RPEL)

## JShell Agenda

1) Introduction to the JShell .....	2
2) Getting Started with JShell .....	3
3) Getting Help from the JShell .....	7
4) Understanding JShell Snippets .....	16
5) Editing and Navigating Code Snippets .....	21
6) Working with JShell Variables .....	23
7) Working with JShell Methods .....	28
8) Using An External Editor with JShell .....	33
9) Using classes, interfaces and enum with JShell .....	35
10) Loading and Saving Snippets in JShell .....	37
11) Using Jar Files in the JShell .....	41
12) How to customize JShell Startup .....	43
13) Shortcuts and Auto-Completion of Commands .....	47



---

# UNIT 1: Introduction to the JShell

Jshell is also known as interactive console.

JShell is Java's own REPL Tool.

REPL means → Read, Evaluate, Print and Loop

By using this tool we can execute Java code snippets and we can get immediate results.

For beginners it is very good to start programming in fun way.

By using this Jshell we can test and execute Java expressions, statements, methods, classes etc. It is useful for testing small code snippets very quickly, which can be plugged into our main coding based on our requirement.

Prior to Java 9 we cannot execute a single statement, expression, methods without full pledged classes. But in Java 9 with JShell we can execute any small piece of code without having complete class structure.

It is not new thing in Java. It is already there in other languages like Python, Swift, Lisp, Scala, Ruby etc..

Python → IDLE

Apple's Swift Programming Language → Playground

## Limitations of JShell:

1. JShell is not meant for Main Coding. We can use just to test small coding snippets, which can be used in our Main Coding.
2. JShell is not replacement of Regular Java IDEs like Eclipse, NetBeans etc
3. It is not that much impressed feature. All other languages like Python, LISP, Scala, Ruby, Swift etc are already having this REPL tools

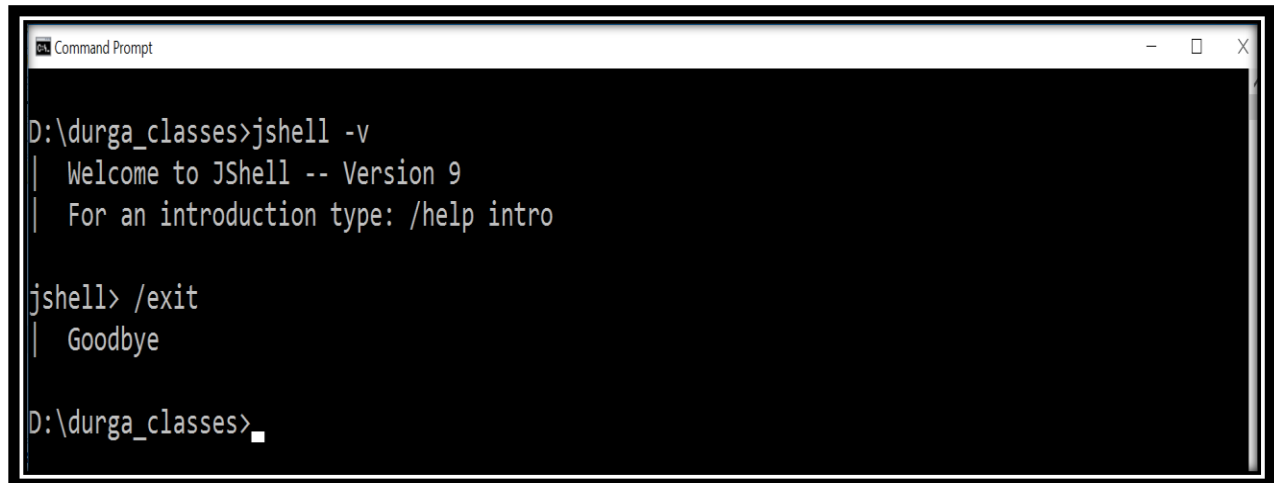


## UNIT-2: Getting Started with JShell

### Starting and Stopping JShell:

Open the jshell from the command prompt in verbose mode

jshell -v



```
Command Prompt
D:\durga_classes>jshell -v
| Welcome to JShell -- Version 9
| For an introduction type: /help intro

jshell> /exit
| Goodbye

D:\durga_classes>_
```

```
D:\durga_classes>jshell -v
| Welcome to JShell -- Version 9
| For an introduction type: /help intro
```

### How to exit jshell:

```
jshell> /exit
| Goodbye
```

**Note:** Observe the difference b/w with -v and without -v (verbose mode)

```
D:\durga_classes>jshell -v
| Welcome to JShell -- Version 9
| For an introduction type: /help intro
```

**Note:** If any information displaying on the jshell starts with '|', it is the information to the programmer from the jshell

```
jshell> 10+20
$1 ==> 30
| created scratch variable $1 : int
```

```
jshell> 20-30*6/2
$2 ==> -70
| created scratch variable $2 : int
```



```
jshell> System.out.println("DURGASOFT")
DURGASOFT
```

Here if we observe the output not starts with | because it is not information from the Jshell.

**Note:** Terminating semicolons are automatically added to the end of complete snippet by JShell if not entered. .

```
jshell> Math.sqrt(4)
$4 ==> 2.0
| created scratch variable $4 : double
```

```
jshell> Math.max(10,20)
$5 ==> 20
| created scratch variable $5 : int
```

```
jshell> Math.random()
$6 ==> 0.6956946870985563
| created scratch variable $6 : double
```

```
jshell> Math.random()
$7 ==> 0.3657412865477785
| created scratch variable $7 : double
```

```
jshell> Math.random()
$8 ==> 0.8828801968574324
| created scratch variable $8 : double
```

**Note:** We are not required to import *Java.lang* package, because by default available.

### Can you check whether the following will work or not?

```
jshell> ArrayList<String> l = new ArrayList<String>();
l ==> []
| created variable l : ArrayList<String>
```

**Note:** The following packages are by default available to the Jshell and we are not required to import. We can check with /imports command

```
jshell> /imports
| import Java.io.*
| import Java.math.*
| import Java.net.*
| import Java.nio.file.*
| import Java.util.*
| import Java.util.concurrent.*
| import Java.util.function.*
| import Java.util.prefs.*
| import Java.util.regex.*
| import Java.util.stream.*
```



```
jshell> ArrayList<String> l=new ArrayList<String>();
l ==> []

jshell> l.add("Sunny");l.add("Bunny");l.add("Chinny");
$2 ==> true
$3 ==> true
$4 ==> true

jshell> l
l ==> [Sunny, Bunny, Chinny]

jshell> l.isEmpty()
$6 ==> false

jshell> l.get(2)
$7 ==> "Chinny"

jshell> l.get(10)
| Java.lang.IndexOutOfBoundsException thrown: Index 10 out-of-bounds for length 3

jshell> l.size()
$9 ==> 3

jshell> if(l.isEmpty()) System.out.println("Empty");else System.out.println("Not Empty");
Not Empty

jshell> for(int i =0;i<10;i=i+2)System.out.println(i)
0
2
4
6
8

Note: Interlly jshell having Java compiler which is responsible to check syntax.If any violation we will get Compile time error which is exactly same as normal compile time errors.

jshell> System.out.println(x+y)
| Error:
| cannot find symbol
| symbol: variable x
| System.out.println(x+y)
|               ^
| Error:
| cannot find symbol
| symbol: variable y
| System.out.println(x+y)
jshell> Sytsem.out.println("Durga")
| Error:
| package Sytsem does not exist
```



```
| System.out.println("Durga")  
| ^-----^
```

**Note:** In our program if there is any chance of getting checked exceptions compulsory we required to handle either by try-catch or by throws keyword. Otherwise we will get Compile time error.

**Eg:**

```
1) import java.io.*;  
2) class Test  
3) {  
4)     public static void main(String[] args)  
5)     {  
6)         PrintWriter pw=new PrintWriter("abc.txt");  
7)         pw.println("Hello");  
8)     }  
9) }
```

D:\durga\_classes>Javac Test.Java

Test.Java:6: error: unreported exception FileNotFoundException; must be caught or declared to be thrown

```
PrintWriter pw=new PrintWriter("abc.txt");
```

But in the case of Jshell, jshell itself will takes care of these and we are not required to use try-catch or throws. Thanks to Jshell.

```
jshell> PrintWriter pw=new PrintWriter("abc.txt");pw.println("Hello");pw.flush();  
pw ==> Java.io.PrintWriter@e25b2fe
```

## Conclusions:

1. From the jshell we can execute any expression, any Java statement.
2. Most of the packages are not required to import to the Jshell because by default already available to the jshell.
3. Internally jshell use Java compiler to check syntaxes
4. If we are not handling any checked exceptions we won't get any compile time errors, because jshell will takes care.



---

## **UNIT - 3: Getting Help from the JShell**

If You Cry For Help....JShell will provide everything.

JShell can provide complete information about available commands with full documentation, and how to use each command and what are various options are available etc..

### **Agenda:**

1) **To know list of options allowed with jshell:**

Type `jshell --help` from normal command prompt

2) **To know the version of jshell:**

Type `jshell --version` from normal command prompt

3) **To know introduction of jshell:**

`jshell> /help intro`

4) **For List of commands:**

type `/help` from jshell

5) **To get information about a particular command:**

`jshell>/help commandname`

6) **To get just names of all commands without any description:**

just type `/` followed by tab

7) **To know the list of options available for a command**

`jshell>/command - tab`

`jshell> /list -`

`-all -history -start`



## 1) To know list of options allowed with jshell

Type `jshell --help` from normal command prompt

```
D:\durga_classes>jshell --help
```

Usage: `jshell <options> <load files>`

where possible options include:

- `--class-path <path>` Specify where to find user class files
- `--module-path <path>` Specify where to find application modules
- `--add-modules <module>(<module>)*`  
Specify modules to resolve, or all modules on the module path if `<module>` is `ALL-MODULE-PATHs`
- `--startup <file>` One run replacement for the start-up definitions
- `--no-startup` Do not run the start-up definitions
- `--feedback <mode>` Specify the initial feedback mode. The mode may be predefined (silent, concise, normal, or verbose) or previously user-defined
- `-q` Quiet feedback. Same as: `--feedback concise`
- `-s` Really quiet feedback. Same as: `--feedback silent`
- `-v` Verbose feedback. Same as: `--feedback verbose`
- `-J<flag>` Pass `<flag>` directly to the runtime system.  
Use one `-J` for each runtime flag or flag argument
- `-R<flag>` Pass `<flag>` to the remote runtime system.  
Use one `-R` for each remote flag or flag argument
- `-C<flag>` Pass `<flag>` to the compiler.  
Use one `-C` for each compiler flag or flag argument
- `--version` Print version information and exit
- `--show-version` Print version information and continue
- `--help` Print this synopsis of standard options and exit
- `--help-extra, -X` Print help on non-standard options and exit

## 2) To know the version of jshell

Type `jshell --version` from normal command prompt

```
D:\durga_classes>jshell --version
jshell 9
```





## 3) To know introduction of jshell

jshell> /help intro

| intro

| The jshell tool allows you to execute Java code, getting immediate results.

| You can enter a Java definition (variable, method, class, etc), like: `int x = 8`

| or a Java expression, like: `x + x`

| or a Java statement or import.

| These little chunks of Java code are called 'snippets'.

| There are also jshell commands that allow you to understand and

| control what you are doing, like: `/list`

| For a list of commands: `/help`

## 4) For List of commands

type `/help` from jshell

jshell> /help

| Type a Java language expression, statement, or declaration.

| Or type one of the following commands:

| `/list [<name or id>|-all|-start]`

| list the source you have typed

| `/edit <name or id>`

| edit a source entry referenced by name or id

| `/drop <name or id>`

| delete a source entry referenced by name or id

| `/save [-all|-history|-start] <file>`

| Save snippet source to a file.

| `/open <file>`

| open a file as source input

| `/vars [<name or id>|-all|-start]`

| list the declared variables and their values

| `/methods [<name or id>|-all|-start]`

| list the declared methods and their signatures

| `/types [<name or id>|-all|-start]`

| list the declared types

| `/imports`

| list the imported items

| `/exit`

| exit jshell

| `/env [-class-path <path>] [-module-path <path>] [-add-modules <modules>] ...`



```
| view or change the evaluation context
| /reset [-class-path <path>] [-module-path <path>] [-add-modules <modules>]...
| reset jshell
| /reload [-restore] [-quiet] [-class-path <path>] [-module-path <path>]...
| reset and replay relevant history -- current or previous (-restore)
| /history
| history of what you have typed
| /help [<command>|<subject>]
| get information about jshell
| /set editor|start|feedback|mode|prompt|truncation|format ...
| set jshell configuration information
| /? [<command>|<subject>]
| get information about jshell
| /!
| re-run last snippet
| /<id>
| re-run snippet by id
| /-<n>
| re-run n-th previous snippet
|
| For more information type '/help' followed by the name of a
| command or a subject.
| For example '/help /list' or '/help intro'.
|
| Subjects:
|
| intro
| an introduction to the jshell tool
| shortcuts
| a description of keystrokes for snippet and command completion,
| information access, and automatic code generation
| context
| the evaluation context options for /env /reload and /reset
```

## **5) To get information about a particular command**

```
jshell>/help commandname
```

```
jshell> /help list
```

```
| /list
|
| Show the source of snippets, prefaced with the snippet id.
|
| /list
| List the currently active snippets of code that you typed or read with /open
|
| /list -start
```



| List the automatically evaluated start-up snippets

| /list -all

| List all snippets including failed, overwritten, dropped, and start-up

| /list <name>

| List snippets with the specified name (preference for active snippets)

| /list <id>

| List the snippet with the specified snippet id

## **To get Information about methods command**

jshell> /help methods

| /methods

| List the name, parameter types, and return type of jshell methods.

| /methods

| List the name, parameter types, and return type of the current active jshell methods

| /methods <name>

| List jshell methods with the specified name (preference for active methods)

| /methods <id>

| List the jshell method with the specified snippet id

| /methods -start

| List the automatically added start-up jshell methods

| /methods -all

| List all snippets including failed, overwritten, dropped, and start-up

## **6) To get just names of all commands without any description**

just type / followed by tab

jshell> /

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

<press tab again to see synopsis>



---

**If we press tab again then we will get one line synopsis for every command:**

**jshell> /**

**/!**

re-run last snippet

**/-<n>**

re-run n-th previous snippet

**/<id>**

re-run snippet by id

**/?**

get information about jshell

**/drop**

delete a source entry referenced by name or id

**/edit**

edit a source entry referenced by name or id

**/env**

view or change the evaluation context

**/exit**

exit jshell

**/help**

get information about jshell

**/history**

history of what you have typed

**/imports**

list the imported items

**/list**

list the source you have typed

**/methods**

list the declared methods and their signatures

**/open**

open a file as source input

**/reload**

reset and replay relevant history -- current or previous (-restore)

**/reset**



**reset jshell**

**/save**

Save snippet source to a file.

**/set**

set jshell configuration information

**/types**

list the declared types

**/vars**

list the declared variables and their values

<press tab again to see full documentation>

If we press tab again then we can see full documentation of command one by one:

jshell> /

/!

Reevaluate the most recently entered snippet.

<press tab to see next command>

jshell> /

/-<n>

Reevaluate the n-th most recently entered snippet.

<press tab to see next command>

jshell> /

/<id>

Reevaluate the snippet specified by the id.

<press tab to see next command>

## **7) To know the list of options available for a command**

jshell>/command - tab

jshell> /list -

-all -history -start

<press tab again to see synopsis>

jshell> /list -



---

### **If we press tab again then we will get synopsis:**

jshell> /list -  
list the source you have typed

<press tab again to see full documentation>  
jshell> /list -

### **If we press tab again then we will get documentation:**

jshell> /list -  
Show the source of snippets, prefaced with the snippet id.

/list  
List the currently active snippets of code that you typed or read with /open

/list -start  
List the automatically evaluated start-up snippets

/list -all  
List all snippets including failed, overwritten, dropped, and start-up

/list <name>  
List snippets with the specified name (preference for active snippets)

/list <id>  
List the snippet with the specified snippet id



---

# Conclusions:

## 1) To know list of options allowed with jshell

Type `jshell --help` from normal command prompt

## 2) To know the version of jshell

Type `jshell --version` from normal command prompt

## 3) To know introduction of jshell

`jshell> /help intro`

## 4) For List of commands

type `/help` from jshell

## 5) To get information about a particular command

`jshell>/help commandname`

## 6) To get just names of all commands without any description

just type `/` followed by tab

## 7) To know the list of options available for a command

`jshell>/command - tab`

`jshell> /list -`

`-all -history -start`



## UNIT - 4: Understanding JShell Snippets

### What are Coding Snippets?

Everything what allowed in Java is a snippet. It can be Expression, Declaration, Statement, classe, interface, method, variable, import, ... We can use all these as snippets from jshell.

\*\*\*But package declarations are not allowed from the jshell.

```
jshell> System.out.println("Hello")
Hello
```

```
jshell> int x=10
x ==> 10
| created variable x : int
```

```
jshell> 10+20
$3 ==> 30
| created scratch variable $3 : int
```

```
jshell> $3>x
$4 ==> true
| created scratch variable $4 : boolean
```

```
jshell> String s =10
| Error:
| incompatible types: int cannot be converted to java.lang.String
| String s =10;
|      ^^
```

```
jshell> String s= "Durga"
s ==> "Durga"
| created variable s : String
```

```
jshell> public void m1()
...> {
...> System.out.println("hello");
...> }
| created method m1()
```

```
jshell> m1()
hello
```





---

**Note:** We can use `/list` command to list out all snippets stored in the jshell memory with snippet id.

jshell> /list

```
1 : System.out.println("Hello")
2 : int x=10;
3 : 10+20
4 : $3>x
5 : String s= "Durga";
6 : public void m1()
  {
    System.out.println("hello");
  }
7 : m1()
```

The numbers 1,2,3 are snippet id. In the future we can access the snippet with id directly.

**Note:** There are some snippets which will be executed automatically at the time jshell start-up, and these are called start-up snippets. We can also add our own snippets as start-up snippets.

We can list out all start-up snippets with command: `/list -start`

jshell> /list -start

```
s1 : import Java.io.*;
s2 : import Java.math.*;
s3 : import Java.net.*;
s4 : import Java.nio.file.*;
s5 : import Java.util.*;
s6 : import Java.util.concurrent.*;
s7 : import Java.util.function.*;
s8 : import Java.util.prefs.*;
s9 : import Java.util.regex.*;
s10 : import Java.util.stream.*;
```

All these are default imports to the jshell.

We can list out all snippets by the command: `/list -all`

jshell> /list -all

```
s1 : import Java.io.*;
s2 : import Java.math.*;
s3 : import Java.net.*;
s4 : import Java.nio.file.*;
s5 : import Java.util.*;
s6 : import Java.util.concurrent.*;
s7 : import Java.util.function.*;
```



```
s8 : import Java.util.prefs.*;
s9 : import Java.util.regex.*;
s10 : import Java.util.stream.*;
1 : System.out.println("Hello")
2 : int x=10;
3 : 10+20
4 : $3>x
e1 : String s =10;
5 : String s= "Durga";
6 : public void m1()
    {
        System.out.println("hello");
    }
7 : m1()
```

We can access snippets by using id directly.

```
jshell> /list 1
```

```
1 : System.out.println("Hello")
```

```
jshell> /list 1 2
```

```
1 : System.out.println("Hello")
2 : int x=10;
```

```
jshell> /list 1 5
```

```
1 : System.out.println("Hello")
5 : String s= "Durga";
```

We can also access snippets directly by using name. The name can be either variable name, class name, method name etc

```
jshell> /list m1
```

```
6 : public void m1()
    {
        System.out.println("hello");
    }
```

```
jshell> /list x
```

```
2 : int x=10;
```

```
jshell> /list s
```

```
5 : String s= "Durga";
```



---

We can execute snippet directly by using id with the command: /id

```
jshell> /3
10+20
$8 ==> 30
| created scratch variable $8 : int
```

```
jshell> /7
m1()
hello
```

We can use drop command to drop a snippet(Making it inactive)  
We can drop snippet by name or id.

```
jshell> /list

1 : System.out.println("Hello")
2 : int x=10;
3 : 10+20
4 : $3>x
5 : String s= "Durga";
6 : public void m1()
   {
       System.out.println("hello");
   }
7 : m1()
8 : 10+20
9 : m1()
```

```
jshell> /drop $3
| dropped variable $3
```

```
jshell> /list

1 : System.out.println("Hello")
2 : int x=10;
4 : $3>x
5 : String s= "Durga";
6 : public void m1()
   {
       System.out.println("hello");
   }
7 : m1()
8 : 10+20
9 : m1()
```

```
jshell> /4
$3>x
| Error:
```



```
| cannot find symbol  
| symbol: variable $3  
| $3>x  
| ^^
```

## Conclusions:

1. We can use /list command to list out all snippets stored in the jshell memory with snippet id.  
jshell>/list

2. In the future we can access the snippet with id directly without retyping whole snippet.  
jshell>/list id

3. There are some snippets which will be executed automatically at the time jshell start-up, and these are called start-up snippets. We can list out all start-up snippets with command: /list -start

```
jshell> /list -start
```

We can also add our own snippets as start-up snippets.

4. The default start-up snippets are default imports to the jshell.

5. We can list out all snippets by the command: /list -all  
jshell> /list -all

6. We can access snippets by using id directly.

```
jshell> /list 1
```

7. We can access snippets directly by using name. The name can be either variable name, class name, method name etc

```
jshell> /list m1
```

8. We can execute snippet directly by using id with the command: /id

```
jshell> /3
```

9. We can use drop command to drop a snippet (Making it inactive)  
We can drop snippet by name or id.

```
jshell> /drop $3
```

Once we dropped a snippet, we cannot use otherwise we will get compile time error.



## UNIT – 5: Editing and Navigating Code Snippets

We can list all our active snippets with `/list` command and we can list total history of our jshell activities with `/history` command.

```
jshell> /list
```

```
1 : int x=10;
2 : String s="Durga";
3 : System.out.println("Hello");
4 : class Test{}
```

```
jshell> /history
```

```
int x=10;
String s="Durga";
System.out.println("Hello");
class Test{}
/list
/history
```

1. By using down arrow and up arrow we can navigate through history. While navigating we can use left and right arrows to move character by character within the snippet.
2. We can use `Ctrl+A` to move to the beginning of the line and `Ctrl+E` to move to the end of the line.
3. We can use `Alt+B` to move backward by one word and `Alt+F` to move forward by one word.
4. We can use `Delete` key to delete the character at the cursor. We can use `Backspace` to delete character before the cursor.
5. We can use `Ctrl+K` to delete the text from the cursor to the end of line.
6. We can use `Alt+D` to delete the text from the cursor to the end of the word.
7. `Ctrl+W` to delete the text from cursor to the previous white space.
8. `Ctrl+Y` to paste most recently deleted text into the line.
9. `Ctrl+R` to Search backward through history
10. `Ctrl+S` to search forward through history



KEY	ACTION
Up arrow	Moves up one line, backward through history
Down arrow	Moves down one line, forward through history
Left arrow	Moves backward one character
Right arrow	Moves forward one character
Ctrl+A	Moves to the beginning of the line
Ctrl+E	Moves to the end of the line
Alt+B	Moves backward one word
Alt+F	Moves forward one word
Delete	Deletes the character at the cursor
Backspace	Deletes the character before the cursor
Ctrl+K	Deletes the text from the cursor to the end of the line
Alt+D	Deletes the text from the cursor to the end of the word
Ctrl+W	Deletes the text from the cursor to the previous white space.
Ctrl+Y	Pastes the most recently deleted text into the line.
Ctrl+R	Searches backward through history
Ctrl+S	Searches forwards through history



## UNIT – 6: Working with JShell Variables

After completing this JShell Variables session, we can answer the following:

1. What are various types of variables possible in jshell?
2. Is it possible to use scratch variable in our code?
3. Is it possible 2 variables with the same name in JShell?
4. If we are trying to declare a variable with the same name which is already available in JShell then what will happen?
5. How to list out all active variables of jshell?
6. How to list out all active& in-active variables of jshell?
7. How to drop variables in the JShell?
8. What is the difference between print() and printf() methods?

In JShell, there are 2 types of variables

1. Explicit variables
2. Implicit variables or Scratch variables

### Explicit variables:

These variables created by programmer explicitly based on our programming requirement.

Eg:

```
jshell> int x = 10
x ==> 10
| created variable x : int
```

```
jshell> String s = "Durga"
s ==> "Durga"
| created variable s : String
```

The variables x and s provided explicitly by the programmer and hence these are explicit variables.

### Implicit Variables:

Sometimes JShell itself creates variables implicitly to hold temporary values, such type of variables are called Implicit variables.

Eg:

```
jshell> 10+20
$3 ==> 30
| created scratch variable $3 : int
```



```
jshell> 10<20
$4 ==> true
| created scratch variable $4 : boolean
```

The variables \$3 and \$4 are created by JShell and hence these are implicit variables.

Based on requirement we can use these scratch variables also.

```
jshell> $3+40
$5 ==> 70
| created scratch variable $5 : int
```

If we are trying to declare a variable with the same name which is already available then old variable will be replaced with new variable.i.e in JShell, variable overriding is possible. In JShell at a time only one variable is possible with the same name.i.e 2 variables with the same name is not allowed.

```
jshell> String x="DURGASOFT"
x ==> "DURGASOFT"
| replaced variable x : String
| update overwrote variable x : int
```

In the above case,int variable x is replaced with String variable x.

While declaring variables compulsory the types must be matched,otherwise we will get compile time error.

```
jshell> String s1=true
| Error:
| incompatible types: boolean cannot be converted to Java.lang.String
| String s1=true;
|      ^^^
```

```
jshell> String s1="Hello"
s1 ==> "Hello"
| created variable s1 : String
```

Note: By using /vars command we can list out type,name and value of all variables which are created in JShell.

Instead of /vars we can also use /var,/va,/v

```
jshell> /help vars
|
| /vars
|
| List the type, name, and value of jshell variables.
|
| /vars
| List the type, name, and value of the current active jshell variables
```





```
|  
| /vars <name>  
| List jshell variables with the specified name (preference for active variables)  
|  
| /vars <id>  
| List the jshell variable with the specified snippet id  
|  
| /vars -start  
| List the automatically added start-up jshell variables  
|  
| /vars -all  
| List all jshell variables including failed, overwritten, dropped, and start-up
```

### **To List out All Active variables of JShell:**

```
jshell> /vars  
| String s = "Durga"  
| int $3 = 30  
| boolean $4 = true  
| String x = "DURGASOFT"  
| String s1 = "Hello"
```

### **To List out All Variables(both active and not-active):**

```
jshell> /vars -all  
| int x = (not-active)  
| String s = "Durga"  
| int $3 = 30  
| boolean $4 = true  
| String x = "DURGASOFT"  
| String s1 = (not-active)  
| String s1 = "Hello"
```

We can drop a variable by using /drop command

```
jshell> /vars  
| String s = "Durga"  
| int $3 = 30  
| boolean $4 = true  
| String x = "DURGASOFT"  
| String s1 = "Hello"
```

```
jshell> /drop $3  
| dropped variable $3
```



```
jshell> /vars
| String s = "Durga"
| boolean $4 = true
| String x = "DURGASOFT"
| String s1 = "Hello"
```

We can create complex variables also

```
jshell> List<String> heroes=List.of("Ameer","Sharukh","Salman");
heroes ==> [Ameer, Sharukh, Salman]
| created variable heroes : List<String>
```

```
jshell> List<String> heroines=List.of("Katrina","Kareena","Deepika");
heroines ==> [Katrina, Kareena, Deepika]
| created variable heroines : List<String>
```

```
jshell> List<List<String>> l=List.of(heroes,heroines);
l ==> [[Ameer, Sharukh, Salman], [Katrina, Kareena, Deepika]]
| created variable l : List<List<String>>
```

```
jshell> /vars
| String s = "Durga"
| boolean $4 = true
| String x = "DURGASOFT"
| String s1 = "Hello"
| List<String> heroes = [Ameer, Sharukh, Salman]
| List<String> heroines = [Katrina, Kareena, Deepika]
| List<List<String>> l = [[Ameer, Sharukh, Salman], [Katrina,Kareena, Deepika]]
```

## System.out.println() vs System.out.printf() methods:

```
public class PrintStream
{
    public void print(boolean);
    public void print(char);
    public void println(boolean);
    public void println(char);
    public PrintStream printf(String,Object...);
    ....
}
```

System.out.println() method return type is void.

But System.out.printf() method return type is PrintStream object. On that PrintStream object we can call printf() method again.

```
jshell> System.out.println("Hello");
Hello
```

```
jshell> System.out.printf("Hello:%s\n","Durga")
Hello:Durga
```



---

```
$11 ==> Java.io.PrintStream@10bdf5e5  
| created scratch variable $11 : PrintStream
```

```
jshell> $11.printf("Hello")  
Hello$12 ==> Java.io.PrintStream@10bdf5e5  
| created scratch variable $12 : PrintStream
```

```
jshell> /vars  
| String s = "Durga"  
| boolean $4 = true  
| String x = "DURGASOFT"  
| String s1 = "Hello"  
| List<String> heroes = [Ameer, Sharukh, Salman]  
| List<String> heroines = [Katrina, Kareena, Deepika]  
| List<List<String>> l = [[Ameer, Sharukh, Salman],  
[Katrina, Kareena, Deepika]]  
| PrintStream $11 = Java.io.PrintStream@10bdf5e5  
| PrintStream $12 = Java.io.PrintStream@10bdf5e5
```

## FAQs:

1. What are various types of variables possible in jshell?
2. Is it possible to use scratch variable in our code?
3. Is it possible 2 variables with the same name in JShell?
4. If we are trying to declare a variable with the same name which is already available in JShell then what will happen?
5. How to list out all active variables of jshell?
6. How to list out all active& in-active variables of jshell?
7. How to drop variables in the JShell?
8. What is the difference between print() and printf() methods?



## UNIT – 7: Working with JShell Methods

In the JShell we can create our own methods and we can invoke these methods multiple times based on our requirement.

Eg:

```
jshell> public void m1()
...> {
...>   System.out.println("Hello");
...> }
| created method m1()
```

```
jshell> m1()
Hello
```

```
jshell> public void m2()
...> {
...>   System.out.println("New Method");
...> }
| created method m2()
```

```
jshell> m2()
New Method
```

In the JShell there may be a chance of multiple methods with the same name but different argument types, and such type of methods are called overloaded methods. Hence we can declare overloaded methods in the JShell.

```
jshell> public void m1(){}
| created method m1()
```

```
jshell> public void m1(int i){}
| created method m1(int)
```

```
jshell> /methods
| void m1()
| void m1(int)
```

We can list out all methods information by using `/methods` command.

```
jshell> /help methods
|
| /methods
|
| List the name, parameter types, and return type of jshell methods.
|
| /methods
```



List the name, parameter types, and return type of the current active jshell methods

**/methods <name>**

List jshell methods with the specified name (preference for active methods)

**/methods <id>**

List the jshell method with the specified snippet id

**/methods -start**

List the automatically added start-up jshell methods

**/methods -all**

List all snippets including failed, overwritten, dropped, and start-up

```
jshell> /methods
```

```
| void m1()
| void m2()
| void m1(int)
```

If we are trying to declare a method with same signature of already existing method in JShell, then old method will be overridden with new method (even though return types are different).  
i.e. in JShell at a time only one method with same signature is possible.

```
jshell> public void m1(int i){}
```

```
| created method m1(int)
```

```
jshell> public int m1(int i){return 10;}
```

```
| replaced method m1(int)
| update overwrote method m1(int)
```

```
jshell> /methods
```

```
| int m1(int)
```

```
jshell> /methods -all
```

```
| void m1(int)
| int m1(int)
```

In the JShell we can create more complex methods also.

**Eg1:** To print the number of occurrences of specified character in the given String

```
1) 5 : public void charCount(String s, char ch)
2)  {
3)    int count=0;
4)    for(int i =0; i <s.length(); i++)
5)    {
6)        if(s.charAt(i)==ch)
```



```
7)      {  
8)          count++;  
9)      }  
10)     }  
11)     System.out.println("The number of occurrences:"+count);  
12)     }
```

```
jshell> charCount("Hello DurgaSoft",'o')  
The number of occurrences:2
```

```
jshell> charCount("Jajaja",'j')  
The number of occurrences:2
```

## **Eg 2: To print the sum of given integers**

```
1) 8 : public void sum(int... x)  
2)  {  
3)    int total=0;  
4)    for(int x1: x)  
5)    {  
6)        total=total+x1;  
7)    }  
8)    System.out.println("The Sum:"+total);  
9)  }
```

```
jshell> sum(10,20)  
The Sum:30
```

```
jshell> sum(10,20,30,40)  
The Sum:100
```

In JShell, inside method body we can use undeclared variables and methods. But until declaring all dependent variables and methods, we cannot invoke that method.

### **Eg1: Usage of undeclared variable inside method body**

```
jshell> public void m1()  
...> {  
...>   System.out.println(x);  
...> }
```

| created method m1(), however, it cannot be invoked until variable x is declared

```
jshell> m1()  
| attempted to call method m1() which cannot be invoked until variable x is declared
```

```
jshell> int x=10  
x ==> 10
```



---

```
| created variable x : int
| update modified method m1()
```

```
jshell> m1()
10
```

## **Eg 2: Usage of undeclared method inside method body**

```
jshell> public void m1()
...> {
...>   m2();
...> }
```

| created method m1(), however, it cannot be invoked until method m2() is declared

```
jshell> m1()
| attempted to call method m1() which cannot be invoked until method m2() is declared
```

```
jshell> public void m2()
...> {
...>   System.out.println("Hello DURGASOFT");
...> }
```

| created method m2()  
| update modified method m1()

```
jshell> m1()
Hello DURGASOFT
```

```
jshell> m2()
Hello DURGASOFT
```

We can drop methods by name with /drop command. If multiple methods with the same name then we should drop by snippet id.

```
jshell> public void m1(){}
| created method m1()
```

```
jshell> public void m1(int i){}
| created method m1(int)
```

```
jshell> public void m2(){}
| created method m2()
```

```
jshell> public void m3(){}
| created method m3()
```

```
jshell> /methods
| void m1()
| void m1(int)
| void m2()
```

```
| void m3()
```

```
jshell> /drop m3
```

| dropped method m3()

```
jshell> /methods
```

```
| void m1()
```

```
| void m1(int)
```

```
| void m2()
```

```
jshell> /drop m1
```

**| The argument references more than one import, variable, method, or class.**

**Use one of:**

```
| /drop 1 : public void m1(){},
```

```
| /drop 2 : public void m1(int i){}
```

```
jshell> /methods
```

```
| void m1()
```

```
| void m1(int)
```

```
void m2()
```

```
jshell> /list
```

```
1 : public void m1(){}
```

```
2 : public void m1(int i){}
```

```
3 : public void m2(){}
```

```
jshell> /drop 2
```

```
| dropped method m1(int)
```

```
jshell> /methods
```

```
| void m1()
```

```
void m2()
```

## FAQs:

1. Is it possible to declare methods in the JShell?
2. Is it possible to declare multiple methods with the same name in JShell?
3. Is it possible to declare multiple methods with the same signature in JShell?
4. If we are trying to declare a method with the same name which is already there in the JShell, but with different argument types then what will happen?
5. If we are trying to declare a method with the same signature which is already there in the JShell, then what will happen?
6. Inside a method if we are trying to use a variable or method which is not yet declared then what will happen?
7. How to drop methods in JShell?
8. If multiple methods with the same name then how to drop these methods?





## UNIT – 8: Using An External Editor with JShell

It is very difficult to type lengthy code from JShell. To overcome this problem, JShell provides an in-built editor.

We can open inbuilt editor with the command: `/edit`

```
jshell> /edit
```

Diagram (image) of inbuilt-editor

If we are not satisfied with JShell in-built editor, then we can set our own editor to the JShell. For this we have to use `/set editor` command.

Eg:

```
jshell> /set editor "C:\\WINDOWS\\system32\\notepad.exe"
```

```
jshell> /set editor "C:\\Program Files\\EditPlus\\editplus.exe"
```

### How to Set Notepad as editor to JShell:

```
jshell> /set editor "C:\\WINDOWS\\system32\\notepad.exe"
```

```
| Editor set to: C:\\WINDOWS\\system32\\notepad.exe
```

If we type `/edit` automatically Notepad will be opened.

```
jshell> /edit
```

But this way of setting editor is temporary and it is applicable only for current session.

If we want to set current editor as permanent, then we have to use the command

```
jshell> /set editor -retain
```

```
| Editor setting retained: C:\\WINDOWS\\system32\\notepad.exe
```

### How to set EditPlus as editor to JShell:

```
jshell> /set editor "C:\\Program Files\\EditPlus\\editplus.exe"
```

```
| Editor set to: C:\\Program Files\\EditPlus\\editplus.exe
```

If we type `/edit` automatically EditPlus editor will be opened.



---

## **How to set default editor once again:**

We have to type the following command from the jshell

```
jshell> /set editor -default  
| Editor set to: -default
```

## **To make default editor as permanent:**

```
jshell> /set editor -retain  
| Editor setting retained: -default
```

**Note:** It is not recommended to set IntelliJ,Eclipse,NetBeans as JShell editors,because it increases startup time and shutdown time of jshell.

## **FAQs:**

1. How to open default editor of JShell?
2. How to configure our own editor to the JShell?
3. How to configure Notepad as editor to the JShell?
4. How to make our customized editor as permanent editor in the JShell?



## UNIT – 9: Using classes, interfaces and enum with JShell

In the JShell we can declare classes, interfaces, enums also.

We can use `/types` command to list out our created types like classes, interfaces and enums.

```
jshell> class Student{}  
| created class Student
```

```
jshell> interface Interf{}  
| created interface Interf
```

```
jshell> enum Colors{}  
| created enum Colors
```

```
jshell> /types  
| class Student  
| interface Interf  
| enum Colors
```

But recommended to use editor to type lengthy classes, interfaces and enums.

```
1) 1 : public class Student  
2) {  
3)     private String name;  
4)     private int rollno;  
5)     Student(String name,int rollno)  
6)     {  
7)         this.name=name;  
8)         this.rollno=rollno;  
9)     }  
10)    public String getName()  
11)    {  
12)        return name;  
13)    }  
14)    public int getRollno()  
15)    {  
16)        return rollno;  
17)    }  
18) }
```

```
jshell> /edit  
| created class Student
```

```
jshell> /types  
| class Student
```

```
jshell> Student s=new Student("Durga",101);  
s ==> Student@754ba872
```



| created variable s : Student

```
jshell> s.getName()
```

```
$3 ==> "Durga"
```

| created scratch variable \$3 : String

```
jshell> s.getRollno()
```

```
$4 ==> 101
```

| created scratch variable \$4 : int

```
1) public interface Interf
2) {
3)     public static void m1()
4)     {
5)         System.out.println("interface static method");
6)     }
7) }
8) enum Beer
9) {
10)    KF("Sour"),KO("Bitter"),RC("Salty");
11)    String taste;
12)    Beer(String taste)
13)    {
14)        this.taste=taste;
15)    }
16)    public String getTaste()
17)    {
18)        return taste;
19)    }
20) }
```

```
jshell> /edit
```

| created interface Interf

| created enum Beer

```
jshell> Interf.m1()
```

```
interface static method
```

```
jshell> Beer.KF.getTaste()
```

```
$8 ==> "Sour"
```

| created scratch variable \$8 : String



## UNIT – 10: Loading and Saving Snippets in JShell

We can load and save snippets from the file.

Assume all our required snippets are available in `mysnippets.jsh`. This file can be with any extension like `.txt`, But recommended to use `.jsh`.

### `mysnippets.jsh`:

```
String s="Durga";
public void m1()
{
    System.out.println("method defined in the file");
}
int x=10;
```

We can load all snippets of this file from the JShell with `/open` command as follows.

```
jshell> /list
```

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

```
jshell> /list
```

```
1 : String s="Durga";
2 : public void m1()
  {
    System.out.println("method defined in the file");
  }
3 : int x=10;
```

Once we loaded snippets, we can use these loaded snippets based on our requirement.

```
jshell> m1()
method defined in the file
```

```
jshell> s
s ==> "Durga"
| value of s : String
```

```
jshell> x
x ==> 10
| value of x : int
```



## Saving JShell snippets to the file:

We can save JShell snippets to the file with /save command.

```
jshell> /help save
|
| /save <file>
|   Save the source of current active snippets to the file.
|
| /save -all <file>
|   Save the source of all snippets to the file.
|   Includes source including overwritten, failed, and start-up code.
|
| /save -history <file>
|   Save the sequential history of all commands and snippets entered since jshell was launched.
|
| /save -start <file>
|   Save the current start-up definitions to the file.
```

**Note:** If the specified file is not available then this save command itself will create that file.

```
jshell> /save active.jsh
```

```
jshell> /save -all all.jsh
```

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

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

```
jshell> /ex
| Goodbye
```

```
D:\>type active.jsh
int x=10;
x
System.out.println("Hello");
public void m1(){}
```

```
D:\>type start.jsh
import Java.io.*;
import Java.math.*;
import Java.net.*;
import Java.nio.file.*;
import Java.util.*;
import Java.util.concurrent.*;
import Java.util.function.*;
import Java.util.prefs.*;
import Java.util.regex.*;
```



---

```
import java.util.stream.*;
```

**Note:** By default, all files will be created in the current working directory. If we want in some other location then we have to use absolute path (Full Path).

```
jshell> /save D:\\durga_classes\\active.jsh
```

## **How to Reload Previous state (session) of JShell:**

We can reload the previous session with the `/reload` command so that all snippets of the previous session will be available in the current session.

```
jshell> /reload -restore
```

**Eg:**

```
jshell> int x=10
x ==> 10
| created variable x : int
```

```
jshell> 10+20
$2 ==> 30
| created scratch variable $2 : int
```

```
jshell> System.out.println("Hello");
Hello
```

```
jshell> /list
```

```
1 : int x=10;
2 : 10+20
3 : System.out.println("Hello");
```

```
jshell> /exit
| Goodbye
```

```
D:\>jshell -v
| Welcome to JShell -- Version 9
| For an introduction type: /help intro
```

```
jshell> /reload -restore
| Restarting and restoring from previous state.
-: int x=10;
-: 10+20
-: System.out.println("Hello");
Hello
```

```
jshell> /list
```



```
1 : int x=10;  
2 : 10+20  
3 : System.out.println("Hello");
```

## How to reset JShell State:

We can reset JShell state by using /reset command.

```
jshell> /help reset  
|  
| /reset  
|  
| Reset the jshell tool code and execution state:  
| * All entered code is lost.  
| * Start-up code is re-executed.  
| * The execution state is restarted.  
| Tool settings are maintained, as set with: /set ...  
| Save any work before using this command.
```

Eg:

```
jshell> /list
```

```
1 : int x=10;  
2 : 10+20  
3 : System.out.println("Hello");
```

```
jshell> /reset  
| Resetting state.
```

```
jshell> /list
```





# UNIT – 11: Using Jar Files in the JShell

It is very easy to use external jar files in the jshell. We can add Jar files to the JShell in two ways.

1. From the Command Prompt
2. From the JShell Itself

## 1. Adding Jar File to the JShell from Command Prompt:

We have to open jshell with --class-path option.

```
D:\>jshell -v --class-path C:\oracle\app\oracle\product\11.2.0\server\jdbc\lib\ojdbc6.jar
```

### Demo Program to get all employees information from oracle database:

#### mynippets.jsh:

```
1) import java.sql.*;
2) public void getEmpInfo() throws Exception
3) {
4)     Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE",
        "scott","tiger");
5)     Statement st=con.createStatement();
6)     ResultSet rs=st.executeQuery("select * from employees");
7)     while(rs.next())
8)     {
9)         System.out.println(rs.getInt(1)+".." +rs.getString(2)+".." +rs.getDouble(3)+".." +rs.getStr
        ing(4));
10)    }
11)    con.close();
12) }
```

#### DaTabase info:

```
1) create Table employees(eno number,ename varchar2(10),esal number(10,2),eaddr varchar
    2(10));
2) insert into employees values(100,'Sunny',1000,'Mumbai');
3) insert into employees values(200,'Bunny',2000,'Hyd');
4) insert into employees values(300,'Chinny',3000,'Hyd');
5) insert into employees values(400,'Vinny',4000,'Delhi');
```

```
D:\>jshell -v --class-path C:\oracle\app\oracle\product\11.2.0\server\jdbc\lib\ojdbc6.jar
```

```
jshell> /open mynippets.jsh
jshell> getEmpInfo()
100..Sunny..1000.0..Mumbai
```



---

200..Bunny..2000.0..Hyd  
300..Chinny..3000.0..Hyd  
400..Vinny..4000.0..Delhi

## 2.Adding Jar File to the JShell from JShell itself:

We can add External Jars to the jshell from the Jshell itself with /env command.

```
jshell> /env --class-path C:\oracle\app\oracle\product\11.2.0\server\jdbc\lib\ojdbc6.jar  
| Setting new options and restoring state.
```

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

```
jshell> getEmpInfo()  
100..Sunny..1000.0..Mumbai  
200..Bunny..2000.0..Hyd  
300..Chinny..3000.0..Hyd  
400..Vinny..4000.0..Delhi
```

**Note:** Internally JShell will use environment variable CLASSPATH if we are not setting CLASSPATH explicitly.



## UNIT – 12: How to customize JShell Startup

By default the following snippets will be executed at the time of JShell Startup.

```
jshell> /list -start
```

```
s1 : import Java.io.*;
s2 : import Java.math.*;
s3 : import Java.net.*;
s4 : import Java.nio.file.*;
s5 : import Java.util.*;
s6 : import Java.util.concurrent.*;
s7 : import Java.util.function.*;
s8 : import Java.util.prefs.*;
s9 : import Java.util.regex.*;
s10 : import Java.util.stream.*;
```

We can customize these start-up snippets based on our requirement.  
Assume our required start-up snippets are available in myStartup.jsh.

mystartup.jsh:

```
int x =10;
String s="DURGA";
System.out.println("Hello Durga Welcome to JShell");
```

To provide these snippets as startup snippets we have to open JShell as follows

```
D:\>jshell -v --startup mystartup.jsh
Hello Durga Welcome to JShell
| Welcome to JShell -- Version 9
| For an introduction type: /help intro
```

```
jshell> /list -start
```

```
s1 : int x =10;
s2 : String s="DURGA";
s3 : System.out.println("Hello Durga Welcome to JShell");
```

Note: if we want DEFAULT import start-up snippets also then we have to open JShell as follows.

```
D:\>jshell -v --startup DEFAULT mystartup.jsh
Hello Durga Welcome to JShell
| Welcome to JShell -- Version 9
| For an introduction type: /help intro
```

```
jshell> /list
```



```
1 : int x =10;
2 : String s="DURGA";
3 : System.out.println("Hello Durga Welcome to JShell");
```

jshell> /list -start

```
s1 : import Java.io.*;
s2 : import Java.math.*;
s3 : import Java.net.*;
s4 : import Java.nio.file.*;
s5 : import Java.util.*;
s6 : import Java.util.concurrent.*;
s7 : import Java.util.function.*;
s8 : import Java.util.prefs.*;
s9 : import Java.util.regex.*;
s10 : import Java.util.stream.*;
```

**Note:** To import all JAVASE packages (almost around 173 packages) at the time of startup we have to open JShell as follows.

```
D:\>jshell -v --startup JAVASE
```

jshell> /list

jshell> /list -start

```
s1 : import Java.applet.*;
s2 : import Java.awt.*;
s3 : import Java.awt.color.*;
..
s172 : import org.xml.sax.ext.*;
s173 : import org.xml.sax.helpers.*;
```

**Note:** In addition to JAVASE, to provide our own snippets we have to open JShell as follows

```
D:\>jshell -v --startup JAVASE mystartup.jsh
Hello Durga Welcome to JShell
| Welcome to JShell -- Version 9
| For an introduction type: /help intro
```

jshell> /list

```
1 : int x =10;
2 : String s="DURGA";
3 : System.out.println("Hello Durga Welcome to JShell");
```

jshell> /list -start

```
s1 : import Java.applet.*;
```



```
s2 : import Java.awt.*;
s3 : import Java.awt.color.*;
..
s172 : import org.xml.sax.ext.*;
s173 : import org.xml.sax.helpers.*;
```

### **Q. What is the difference between the following?**

1. jshell -v
2. jshell -v --startup mystartup.jsh
3. jshell -v --startup DEFAULT mystartup.jsh
4. jshell -v --startup JAVASE
5. jshell -v --startup JAVASE mystartup.jsh

### **Need of PRINTING Option at the startup:**

Usually we can use `System.out.print()` or `System.out.println()` methods to print some statements to the console. If we use PRINTING Option then several overloaded `print()` and `println()` methods will be provided at the time of startup and these internally call `System.out.print()` and `System.out.println()` methods.

Hence to print statements to the console just we can use `print()` or `println()` methods directly instead of using `System.out.print()` or `System.out.println()` methods.

```
D:\>jshell -v --startup PRINTING
```

```
jshell> /list -start
```

```
s1 : void print(boolean b) { System.out.print(b); }
s2 : void print(char c) { System.out.print(c); }
s3 : void print(int i) { System.out.print(i); }
s4 : void print(long l) { System.out.print(l); }
s5 : void print(float f) { System.out.print(f); }
s6 : void print(double d) { System.out.print(d); }
s7 : void print(char s[]) { System.out.print(s); }
s8 : void print(String s) { System.out.print(s); }
s9 : void print(Object obj) { System.out.print(obj); }
s10 : void println() { System.out.println(); }
s11 : void println(boolean b) { System.out.println(b); }
s12 : void println(char c) { System.out.println(c); }
s13 : void println(int i) { System.out.println(i); }
s14 : void println(long l) { System.out.println(l); }
s15 : void println(float f) { System.out.println(f); }
s16 : void println(double d) { System.out.println(d); }
s17 : void println(char s[]) { System.out.println(s); }
s18 : void println(String s) { System.out.println(s); }
s19 : void println(Object obj) { System.out.println(obj); }
```



```
s20 : void printf(Locale l, String format, Object... args) { System.out.printf(l, format, args);  
}
```

```
s21 : void printf(String format, Object... args) { System.out.printf(format, args); }
```

Now onwards, to print some statements to the console directly we can use `print()` and `println()` methods.

```
jshell> print("Hello");  
Hello  
jshell> print(10.5)  
10.5
```

**Note:**

1. Total 21 overloaded `print()`, `println()` and `printf()` methods provided because of PRINTING shortcut.
2. Whenever we are using PRINTING shortcut, then DEFAULT imports won't come. Hence, to get DEFAULT imports and PRINTING shortcut simultaneously, we have to open JShell as follows.

```
D:\>jshell -v --startup DEFAULT PRINTING
```

**Note:**

Various allowed options with `--startup` are :

1. DEFAULT
2. JAVASE
3. PRINTING



---

## **UNIT – 13: Shortcuts and Auto-Completion of Commands**

### **Shortcut for Creating Variables:**

Just type the value on the JShell and then "Shift+Tab followed by v" then complete variable declaration code will be generated we have to provide only name of the variable.

```
jshell> "Durga" // just press "Shift+Tab followed by v"
jshell> String s= "Durga"
We have to provide only name s
```

```
jshell> 10.5 // just press "Shift+Tab followed by v"
jshell> double d = 10.5
```

### **Shortcut for auto-import:**

just type class or interface name on the JShell and press "Shift+Tab followed by i". Then we will get options for import.

```
jshell> Connection // press "Shift+Tab followed by i"
0: Do nothing
1: import: com.sun.jdi.connect.spi.Connection
2: import: Java.sql.Connection
Choice: //enter 2
Imported: Java.sql.Connection
```

```
jshell> /imports
| import Java.io.*
| import Java.math.*
| import Java.net.*
| import Java.nio.file.*
| import Java.util.*
| import Java.util.concurrent.*
| import Java.util.function.*
| import Java.util.prefs.*
| import Java.util.regex.*
| import Java.util.stream.*
| import Java.sql.Connection
```



## Auto Completion commands :

### 1. To get all static members of the class:

```
jshell>classsname.<Tab>
```

Eg:

```
jshell> String.<Tab>  
CASE_INSENSITIVE_ORDER class copyValueOf(  
format( join( valueOf(
```

### 2. To get all instance members of class:

```
jshell>objectreference.<Tab>
```

```
jshell> String s="Durga";  
jshell> s.<Tab>  
charAt( chars() codePointAt( codePointBefore(  
codePointCount( codePoints() compareTo( compareToIgnoreCase(  
concat( contains( contentEquals( endsWith(  
equals( equalsIgnoreCase( getBytes( getChars(  
getClass() hashCode() indexOf( intern()  
isEmpty() lastIndexOf( length() matches(  
notify() notifyAll() offsetByCodePoints( regionMatches(  
replace( replaceAll( replaceFirst( split(  
startsWith( subSequence( substring( toCharArray()  
toLowerCase( toString() toUpperCase( trim()
```

### 3. To get signature and documentation of a method:

```
jshell> classname.methodname(<Tab>  
jshell> objectreference.methodname(<Tab>
```

```
jshell> s.sub<Tab>  
subSequence( substring(
```

```
jshell> s.substring(  
substring(
```

```
jshell> s.substring(<Tab>  
Signatures:  
String String.substring(int beginIndex)  
String String.substring(int beginIndex, int endIndex)
```

```
<press Tab again to see documentation>  
jshell> s.substring(<Tab>
```





**String** `String.substring(int beginIndex)`

Returns a string that is a substring of this string. The substring begins with the character at the specified index and extends to the end of this string.

**Examples:**

"unhappy".substring(2) returns "happy"

"Harbison".substring(3) returns "bison"

"emptiness".substring(9) returns "" (an empty string)

**Parameters:**

`beginIndex` - the beginning index, inclusive.

**Returns:**

the specified substring.

**Note:** Even this <Tab> short cut applicable for our own classes and methods also.

```
jshell> public void m1(int...x){}  
| created method m1(int...)
```

```
jshell> m1(<Tab>  
m1(  
|
```

```
jshell> m1(<Tab>  
Signatures:  
void m1(int... x)
```