Java 1.9 Features

**Date 03/02/2023**

# Overview of Java 1.9 (21st Sept 2017)

Features of Java 1.9 as below,

1. JShell
2. JPMS (Java Platform Module System)
3. JLINK (Java Linker)
4. HTTP/2 client API
5. Process API Updates (Rank 1 feature of Java 1.9)
6. Private Methods inside Interface
7. Try with Resources Enhancements
8. Factory methods to create Unmodifiable Collections
9. Stream API enhancements
10. <> operator enhancement (applicable for anonymous inner class) – The operator was introduced in Java 1.7
11. Safe Varargs Annotations
12. G1GC (Garbage First Garbage Collector) which is introduced in Java 1.6

# Private Methods inside Interface

Every method present inside the Interface is **public** and **abstract** by default. Likewise, the variable present inside the interface is **public static final** by default whether declared or not. But this rule is applicable until java **1.7 v**.

From, java **1.8 v** onwards, inside interface **default** and **static** methods also allowed.

From, java **1.9 v** onwards, inside interface **private** methods are also allowed.

## Need of Default Method & Static Method inside Interface

Until java 1.7v, it is not possible to add new method inside existing interface without affecting the interface i.e., few methods already present inside interface and those were implemented by 1000 implementation classes.

After analyzing this problem, the Open JDK people have introduced the default method inside interface. Using the default methods, we can add new methods without affecting the functionality of the interface and the corresponding implemented classes.

If the functionality is no way related to object, then why to have that functionality inside classes. So, we can have those functionality inside Interface i.e., why the static methods are allowed inside the Interface.

## Need of Private Method inside Interface

If several default methods present inside the Interface have common functionality, then there may be a chance of code redundancy.

In java 1.9, we can have private (instance & static) methods inside the interface. If we want to achieve code reusability without affecting the functionality of the interface and the implemented classes. Then, we should go for Private Method.

Example,

interface **Interf** {

default void logInfo (String message) {

Step1: Connect to DB

Step2: Log Info Message

Step3: Close the DB connection

}

default void logWarn (String message) {

Step1: Connect to DB

Step2: Log Warn Message

Step3: Close the DB connection

}

default void logError (String message) {

Step1: Connect to DB

Step2: Log Error Message

Step3: Close the DB connection

}

}

In all the above log methods having some common code which increases the length of the code and reduces the code readability. Open JDK people address this issue and provided the solution as introduction of private methods inside Interface. We can separate the common code inside a private method and can call the private method from every default method.

interface **Interf** {

default void logInfo (String message) {

log(message, INFO);

}

default void logWarn (String message) {

log(message, WARN);

}

default void logError (String message) {

log(message, ERROR);

}

Private void log(String msg, String logLevel) {

Step1: Connect to DB

Step2: Log Message with provided log level

Step3: Close the DB connection

}

}

Private static methods will provide code reusability for public static methods.

Interface **Interf** {

public static void m1 () {

}

public static void m2 () {

}

private static void m3 () {

SOP (Common functionality method for m1 and m2);

}

}

class **Test** implements **Interf** {

public static void main (String[] args) {

Interf.m1();

Interf.m2();

}

## Advantages of Private Methods Inside Interface

The main advantage of having private methods inside interfaces are,

1. Code reusability
2. We can expose only intended methods to the API clients (Implementation classes). Bcoz, interface private methods are not visible to the implementation class.

**Note**

1. Private methods can’t be abstract and hence it should compulsorily have the method body.
2. Private methods inside interface can be either static or non-static.

## Structure of Interface after Java 1.9

interface **Java9Interface** {

public static final variables

public abstract methods

default methods with implementations

public static methods with implementations

private instance methods with implementations

private static methods with implementations

}

# Try with Resources Enhancements

Until java 1.6v, its highly recommended to write the finally block to close all the resources which are open as part of the try block.

Example,

BufferedReader br = null;

try {

br = new BufferedReader (new FileReader(“abc.txt”));

Risky code;

}

catch (IOException ex) {

Handling code;

}

finally {

if (br!=null) {

br.close();

}

}

Whatever resources we opened as part of try block must be closed in the finally block and it has to be done by programmer explicitly. It leads to complexity if the programmer forgot to close the resources and also the length of the code increases.

## Need of Try with Resources

The main advantage of “try with resources” is the resources which are opened as part of try block will be closed automatically once the control reaches end of the try block either normally or abnormally and hence, we are not required to close explicitly so that complexity of the programming will be reduced. Also, it’s not required to write finally block explicitly and hence length of the code will be reduced and readability will be improved.

The above code snippet written in the java 1.7 version using try with resources format as below,

try (BufferedReader br = new BufferedReader (new FileReader(“abc.txt”))) {

br = new BufferedReader (new FileReader(“abc.txt”));

Risky code;

}

catch (IOException ex) {

Handling code;

}

**Conclusion**

1. We can declare any number of resources but all resources should be separated with; (semicolon)
2. All resources should be AutoCloseable resources. A resource is said to be **AutoCloseable** if and only if the corresponding class implements the **java.lang.AutoCloseable interface** either directly or indirectly.

All DB related, network related and File IO related resources already implemented AutoCloseable interface. Being a programmer, we should aware this point and we are not required to do anything extra.

1. AutoCloseable interface introduced in Java1.7 version and it contains only one method **close ()**.
2. All resource reference variables should be final or effectively final and hence we can’t perform reassignment within try catch block.

try (BufferedReader br = new BufferedReader (new FileReader(“abc.txt”))) {

br = new BufferedReader (new FileReader(“abc.txt”));

1. Until java 1.6v, try should be followed by either catch or finally but from 1.7v onwards we can take only try with resource without catch or finally.

**Problem with Java 7 Try with Resource**

1. The resource reference variables which are created outside of try block cannot be used directly in try with resources.

BufferedReader br = new BufferedReader (new FileReader(“abc.txt”));

try(br) {

//Risky code;

}

The above syntax is invalid in Java 1.8v.

We should create the resource in try block primary list or we should declared with new reference variable in try block. i.e., resource reference variable should be local to try block.

**Solution 1**

try (BufferedReader br = new BufferedReader (new FileReader(“abc.txt”))) {

// It is valid syntax in Java 1.8v

}

**Solution 2**

BufferedReader br = new BufferedReader (new FileReader(“abc.txt”));

try (BufferedReader br1 = br) {

// It is valid syntax in Java 1.8v

}

## Java 1.9v Try with Resource Enhancement

But from Java 1.9v onwards, we can use the resource reference variables which are created outside of try block directly in try block resource list i.e., the resource reference variables need not be local to try block.

BufferedReader br = new BufferedReader (new FileReader(“abc.txt”));

try (br) {

// It is valid syntax in Java 1.9v

}

But make sure resource (br) should be either final or effectively final. Effectively final means we should not perform reassignment.

## Example

Creating the custom resource class by implementing the Autocloseable interface.

class **MyResource** implements **AutoCloseable** {

MyResource () {

SOP (Resource Creation);

}

Public void doProcess () {

SOP (Resource processing);

}

Public void close () {

SOP (Resource processing);

}

}

**Try Catch in java 1.6v style**

MyResource r = null;

try {

r = new MyResource ();

r.doProcess ();

}

catch (Exception e) {

SOP (Handling + e);

}

finally {

if (r!= null) {

r.close();

}

}

**Try Catch in java 1.7v style**

try (MyResource r = new MyResource ()) {

r.doProcess ();

}

catch (Exception e) {

SOP (Handling + e);

}

**Try Catch in java 1.9v style**

MyResource r = new MyResource ();

Try (r) {

r.doProcess ();

}

catch (Exception e) {

SOP (Handling + e);

}

# Diamond Operator Enhancements (Milling Project Coin - JEP 213)

Java (JDK) Enhancement Proposal

## Purpose of Generics

The purpose of Generics as below,

1. To provide Type Safety
2. To resolve Type Casting

Arrays are always type safe whereas Collections are not type safe until java 1.4 version. In order to solve this problem, the generic concept introduced in java 1.5v for Collections.

In the case of Arrays, while retrieve data from Arrays, it always has the same type of data whereas in Collections it can hold any type of data and therefore the data type of collection is Object and we need to type cast it during the time of retrieval of data from Collections. In order to solve this problem, the Type Casting concept introduced in java 1.5v for Collections.

## Generic Version of ArrayList

**ArrayList<String> al = new ArrayList<String> ();**

**Type Safety Achieved**

al.add(“Java”);

al.add(new Integer(10)); -> Invalid

**Type Casting (Not Required)**

String value = al.get(0);

## Difference b/w Generic and Non-Generic Collections

|  |  |
| --- | --- |
| ArrayList al = new ArrayList (); | ArrayList<String> al = new ArrayList<String> (); |
| It is non-generic version of ArrayList | It is generic version of ArrayList |
| For this ArrayList we can add any type of object and hence it’s not type safe. | For this ArrayList we can add only String type of object and hence it is type safe. |
| At the time of retrieval of data, we should perform type casting. | At the time of retrieval of data, we don’t need to perform type casting. |

## Need of <> Operator

As of java 1.5 v, using Generics we need to repeat the type parameters two times while declaring collection as below,

ArrayList<**String**> al = new ArrayList<**String**> ();

To avoid this repetition of type parameter, in java 1.7 v, <> operator was introduced and it is called as **Type Inference**. Using, this we don’t need to declare the type in two places instead we can declare only once as below. The compiler will automatically consider the type.

ArrayList<**String**> al = new ArrayList**< >** ();

But this <> operator is only applicable for normal top level classes but for anonymous classes its not applicable.

The main objective of Diamond operator is to instantiate generic classes very easily. We are not required to specify the type parameter in the constructor explicitly, length of the code will be reduced and readability will be improved.

## <> Operator enhancements

From java1.9v onwards, the <> operator is applicable for anonymous classes as well.

Sometimes we can declare the classes without having the name, such type of classes are called as Anonymous Classes.

Example,

Thread t = new Thread () {  
};

We are creating the child class extending the Thread class without name (anonymous class) and we are creating object for that child class.

Example,

class **MyGenClass**<T> {

T obj;

public MyGenClass (T obj) {

this.obj=obj;

}

public T getObj () {

return obj;

}

public void process () {

SOP (Processing Obj);

}

}

public class **Test** {

public static void main (String[] args) {

MyGenClass<String> c1 = new MyGenClass<>(“Java”) {

public void process () {

SOP (“Processing “+getObj);

}

};

c1.process();

}

}

# SafeVarArgs Annotation Enhancements

To understand the importance of this annotation first we should aware about the var-arg methods and heap pollution problem.

## VarArg Method

Until java 1.4v, we can’t declare a method with variable number of arguments. If there is a change in number of arguments compulsory, we have to define a new method. This approach increases length of the code and reduces readability.

But from java 1.5v onwards, we can declare a method with variable number of arguments, such type of methods is called var-arg methods.

## VarArg Method with Generic Type

If we are using the VarArg method with Generic Type then may be a chance of Heap Pollution. At runtime, if one type variable trying to point to another type value, then there may be a chance of ClassCastException. This problem is called as Heap Pollution.

In our code, if there is chance of heap pollution then compiler will generate warnings.

Example,

import java.util.\*;

public class **Test** {

public static void main (String[] args) {

List<String> al1 = Arrays.asList(“A”, “B”);  
List<String> al2 = Arrays.asList(“A”, “B”);  
m1(al1, al2);

}

public static void m1(List<String>… al) {

Object[] a = al;

a[0] = Arrays.asList(10,20);

**String var = (String)al[0].get(0);**

SOP (var);

}

}

While compilation, we will get the below error

Test.java uses unchecked or unsafe operations

Recompile with -Xlint:unchecked for details

**javac -Xlint:unchecked Test.java**

warning: [unchecked] unchecked generic array creation for Varargs parameter of type List<String>[] m1(al1, al2);

warning: [unchecked] Possible heap pollution from parameterized vararg type List<String>

public static void m1(List<String>… al)

**java Test**

RE: java.lang.ClassCastException:java.base/java.lang.Integer cannot be cast to java.base/java.lang.String

In the above program at runtime, String type variable var is trying to point to Integer type, which causes Heap Pollution and results ClassCastException.

## Need of @SafeVarArgs Annotation

Very few Var-Arg methods causes Heap Pollution, not all the var-arg methods. If we know, that our method won’t cause Heap Pollution, then we can suppress compiler warnings with @SafeVarargs annotation.

Example,

import java.util.\*;

public class **Test** {

public static void main (String[] args) {

List<String> al1 = Arrays.asList(“A”, “B”);  
List<String> al2 = Arrays.asList(“A”, “B”);  
m1(al1, al2);

}

**@SafeVarargs**

public static void m1(List<String>… al) {

for (List<String> all:al) {

SOP (all);

}

}

}

The SafeVarArgs annotation was introduced in the java 1.7 v. And it is available for Constructors, Static Methods, and Final Methods.

From java 1.9 v onwards, it will be applicable for Private Methods as well.

Example,

import java.util.\*;

public Class **Test** {

**@safeVarags – Applicable from Java 1.7v**

public Test (List<String>… al) {

}

**@safeVarags – Applicable from Java 1.7v**

public static void m1 (List<String>… al) {

}

**@safeVarags – Applicable from Java 1.7v**

public final void m2 (List<String>… al) {

}

**@safeVarags – Applicable only from Java 1.9v**

private void m3 (List<String>… al) {

}

}

# Factory Method for Creating Unmodifiable Collections

**Factory methods for collection**, which is introduced as part of **JEP 269**. The JEP stands for JDK enhancement proposal.

Java 9 Collection library includes static factory methods for List, Set and Map interface. These methods are useful to create unmodifiable collections.

Until java 1.8v, Suppose, if we want to create a list of 5 elements, we need to write the following code.

Example,

import java.util.\*;

public class **FactoryMethodsExample** {

public static void main (String[] args) {

List<String> list = new ArrayList<> ();

list.add("Java");

list.add("JavaFX");

list.add("Spring");

list.add("Hibernate");

list.add("JSP");

for (String al: list) {

System.out.println(al);

}

}

In the above code, add method is called repeatedly for each list element, while in Java 9 we can do it in single line of code using factory methods.

## Need for Factory Method

Factory methods are special type of static methods that are used to create unmodifiable instances of collections. It means we can use these methods to create list, set and map of small number of elements.

## List.of()

In Java 9, we can write the above code in very simple manner with the help of **List.of()** factory method.

The List instances created by these methods have the following characteristics:

1. These lists are immutable. Elements cannot be added, removed, or replaced in these lists. Calling any mutator method (i.e., add, addAll, clear, remove, removeAll, replaceAll) will always cause **UnsupportedOperationException** to be thrown.
2. They do not allow null elements. Attempts to add null elements result in **NullPointerException**.
3. They are serializable if all elements are serializable.
4. The order of elements in the list is the same as the order of the provided arguments, or of the elements in the provided array.

Example,

import java.util.\*;

public class **FactoryMethodsExample** {

public static void main (String[] args) {

**List<String> list = List.of("Java","JavaFX","Spring","Hibernate","JSP");**

for (String l:list) {

System.out.println(l);

}

}

}

## Set.of()

Java Set interface provides a **Set.of()** static factory method which is used to create immutable set. The set instance created by this method has the following characteristics.

1. It is immutable
2. No null elements
3. It is serializable if all elements are serializable.
4. No duplicate elements. Set do not allow duplicate elements as well. Any duplicate element passed will result in **IllegalArgumentException**.
5. The iteration order of set elements is unspecified and is subject to change.

The above code is written with Set.of() as below,

Example,

import java.util.\*;

public class **FactoryMethodsExample** {

public static void main (String[] args) {

**Set<String> set = Set.of("Java","JavaFX","Spring","Hibernate","JSP");**

for (String l: set) {

System.out.println(l);

}

}

}

## Map.of()

In Java 9, Map includes **Map.of()** and **Map.ofEntries()** static factory methods that provide a convenient way to create immutable maps.

Map created by these methods has the following characteristics.

1. It is immutable
2. It does not allow null keys and values
3. It is serializable if all keys and values are serializable
4. It rejects duplicate keys at creation time
5. The iteration order of mappings is unspecified and is subject to change.

Example,

import java.util.\*;

public class **FactoryMethodsExample** {

public static void main (String[] args) {

**Map<Integer, String> map = Map.of(101,"JavaFX",102,"Hibernate",103,"Spring MVC");**

for (Map.Entry<Integer, String> m: map.entrySet()) {

System.out.println(m.getKey()+" "+m.getValue());

}

}

}

In Java 9, apart from static **Map.of()** methods, Map interface includes one more static method **Map.ofEntries()**. This method is used to create a map of Map.Entry instances.

Example,

import java.util.\*;

public class **FactoryMethodsExample** {

public static void main(String[] args) {

// Creating Map Entry

Map.Entry<Integer, String> e1 = Map.entry(101, "Java");

Map.Entry<Integer, String> e2 = Map.entry(102, "Spring");

// Creating Map using map entries

Map<Integer, String> map = Map.ofEntries(e1,e2);

// Iterating Map

for(Map.Entry<Integer, String> m : map.entrySet()){

System.out.println(m.getKey()+" "+m.getValue());

}

}

}

## Notes

1. For List and Set interfaces, of(...) method is overloaded to have **0 to 10** parameters and one with var args parameter.
2. For Map interface, of(...) method is overloaded to have **0 to 10** parameters.
3. In case of more than **10** parameters for Map interface, **ofEntries(...)** method can be used accepting var args parameter.

# Stream API Enhancements

If we want to process a group of objects from the collection then we should go for Streams. The main objective of Streams concept is to process the elements of collection with functional programming (Lambda expression).

## Java9 Stream API Enhancements

As part of Java9 enhancements to Stream API, the following methods were introduced.

1. takeWhile ()
2. dropWhile ()
3. Stream.iterate ()
4. Stream.ofNullable ()

**Note:** takeWhile () and dropWhile () are default methods whereas Stream.iterate() and Stream.ofNullable() are static methods.

### takeWhile ()

It is the default method present in the Stream interface.

**default Stream takeWhile (Predicate p)**

It returns the stream of elements that matches the given predicate.

**filter ()** method checks every element of the stream whereas **takeWhile ()** method check until the condition is true. i.e., takeWhile() takes elements from stream as long as predicate returns true. If predicate returns false, at that point onwards remaining elements won’t be processed, i.e., rest of the stream is discarded.

public class **TakeWhile\_Demo1** {

public static void main(String[] args) {

List<Integer> numbers = List.of(2, 4, 8, 1, 6, 9, 10);

// With filter()

List<Integer> numbersAfterFilter = numbers.stream().**filter**(n -> n % 2 == 0).collect(Collectors.toList());

System.out.println(numbersAfterFilter);

// With takeWhile

List<Integer> numbersAfterTakeWhile = numbers.stream().**takeWhile**(n -> n % 2 == 0).collect(Collectors.toList());

System.out.println(numbersAfterTakeWhile);

}}

**Output** [2, 4, 8]

### dropWhile ()

It is the default method present in the Stream interface.

**default Stream dropWhile (Predicate p)**

It is the opposite of the takeWhile () method.

It drops the elements instead of taking them as long as predicate returns true. Once predicate returns false then rest of the Stream will be returned.

Example,

public class **DropWhile\_Demo1** {

public static void main(String[] args) {

List<Integer> numbers = List.of(2, 4, 8, 1, 6, 9, 10);

// With filter()

List<Integer> numbersAfterFilter = numbers.stream().**filter**(n -> n % 2 == 0).collect(Collectors.toList());

System.out.println(numbersAfterFilter);

// With takeWhile

List<Integer> numbersAfterTakeWhile = numbers.stream().**dropWhile**(n -> n % 2 == 0).collect(Collectors.toList());

System.out.println(numbersAfterTakeWhile);

}

}

**Output** [1, 6, 9, 10]

### Stream.iterate ()

It is the static method present in the Stream interface.

**Form 1: iterate () method with 2 arguments**

This method is introduced in Java 8. It takes the initial value and the function that provides the next value.

**public static Stream iterate (T initial, UnaryOperator<T> f)**

Example,

Stream.iterate(1, x-> x+1).forEach(System.out::println);

**Output**

1

2

3

Infinite times

Stream.iterate(1, x-> x+1).limit(5).forEach(System.out::println);

**Output**

1

2

3

4

5

**Form 2: iterate () method with 3 arguments**

The problem with 2 arguments iterate () method is there may be chance of infinite loop. To avoid this, we should use limit method.

To prevent the infinite loop, in Java 9, another version of iterate () method is introduced with 3 arguments. This method is something like for loop.

for (int i=0; i<10; i++) {}

**public static Stream iterate (T initial, Predicate condition check, UnaryOperator<T> f)**

This method takes an initial value, predicate to terminate and function that provides next value.

Example, Stream.iterate(1, x->x<5,x->x+1).forEach(System.out::println);

**Output**

1

2

3

4

### Stream.ofNullable()

This method will check whether the provided element is null or not. If its not null, then this method returns the Stream of that element. If its null then this method returns empty stream.

**public static Stream<T>ofNullable(T t)**

This method is helpful to deal with null values in the Stream. The main advantage of this method is to avoid NullPointerException and null checks everywhere.

Usually, we can use this method in flatMap to handle null values.

Example,

List al = Stream.ofNullable(100).collect(Collectors.toList());

SOP(al);

**Output**

[100]

List al = Stream.ofNullable(null).collect(Collectors.toList());

SOP(al);

**Output**

[]

Example,

public Class **Test** {

public static void main (String[] args) {

List<String> al = new ArrayList<String>();

al.add(“A”);

al.add(“B”);

al.add(null);

al.add(“C”);

al.add(null);

}

}

SOP (al);

List<String> l2 = al.stream().filter(o->o!=null).collect(Collectors.toList());

SOP (l2);

List<String> l3 = al.stream().flatMap(o->Stream.ofNullable(o)).collect(Collectors.toList());

SOP (l3);

**Output**

[A, B, null, C, null]

[A, B, C]

[A, B, C]

public Class **Test** {

public static void main (String[] args) {

Map<String, String> map = new HashMap<>();

map.put(“A”, “Apple”);

map.put(“B”, “Banana”);

map.put(“C”, null);

map.put(“D”, “DragonFruit”);

List<String> al = map.entrySet().stream().map(e->e.getKey()).collect(Collectors.toList());

SOP(al);

List<String> al2 = map.entrySet().stream().flatMap(e->Stream.ofNullable(e.getValue())).collect(Collectors.toList());

SOP(al2);

}

}

**Output**

[A, B, C, D]

[Apple, Banana, DragonFruit]

# Java Platform Module System (Jigsaw project)

Until Java 1.8v, we can develop applications by writing several classes, interfaces and enum. We can place these components inside a package and we can convert these packages into jar files. By placing these jar files in classpath, we can run our applications. An enterprise application can contain 1000s of jar files. A jar file is a file which contains a group of packages and packages contains a group of .class files. We have some issues with jar file creation.

## Intro

To overcome / fix this jar file issue, JDK people came up with new enhancement called as Module. From Java 1.9v onwards, a new construct got introduced which is called as Module. From java9, we can develop applications using module concept.

A module is a group of packages similar to the jar file. But the specialty of module when compared to jar file is, module can contain configuration information inside a special file called as **module-info.java**. Every module should compulsory contains module-info.java, otherwise JVM won’t consider that as module of java9 platform.

Module system is a part of Jigsaw Project. It adds one more abstraction level above packages. In other words, it is a ‘package of Packages’ that makes our code even more reusable.

It is also fine to say that a module is a group of closely related packages, resources and module descriptor(module-info.java) file.

In Java 9, JDK itself is modularized. All classes of Java 9 are grouped into several modules (around 98) like,

java.base

java.sql

java.logging

java.desktop

java.rmi

To get the module information about the Class,

SOP (String.class.getModule());

SOP (Connection.class.getModule());

SOP (ArrayList.class.getModule());

## Need of JPMS

Applications developed using the jar files has following problems and to overcome these issues JPMS is introduced.

1. Unexpected NoClassDefFoundError in middle of program execution.
2. There is no way to specify the jar file dependencies until java 1.8 v. At runtime, if any dependent jar file is missing then there may be chance of NoClassDefFoundError.
3. There may be chance of duplicate versions of .class files available and which may cause the version conflicts and cause abnormal behaviour.
4. We don’t control over the Jar file and all the items available in the jar file is public to everyone.
5. The Jar file development follows monolithic and heavy weight.

## Difference b/w java8 jar and java9 module

|  |  |
| --- | --- |
| **Jar File** | **Module** |
| Jar is group of packages and each package contains several classes. | Module is also a group of packages and each package contains several classes. Module can also contain one special file module-info.java to hold module specific dependencies and configuration information. |
| In Jar file, there is no way to specify dependent jar files information. | For every module we have to maintain a special file module-info.java to specify module dependencies. |
| There is no way to check all jar file dependencies at the beginning only. Hence in the middle of the program execution there may be a chance of **NoClassDefFoundError**. | JVM will check all module dependencies at the beginning only with the help of module-info.java. If any dependent module is missing then JVM won’t start the execution. Hence, there is no chance of **NoClassDefFoundError** in the middle of execution. |
| In the classpath, the order of jar files is important and JVM will always consider from left to right for the required .class files. If multiple jars contain the same .class file then there may be a chance of **version conflicts** and results in abnormal behavior of the application. | In the module path, order is not important. JVM will always check from the dependent module only for the required .class files. Hence, there is **no chance of version conflicts** and abnormal behavior of the application. |
| In jar file there is no mechanism to control access to the packages. Everything present in the jar file is public to everyone. Any person is allowed to access component from the jar file. Hence there may be a chance of **security problems**. | In module, there is a mechanism to control the access to the packages. Only exported packages are visible to other modules. Hence there is **no chance of security problems**. |
| Jar files follow **monolithic structure** and applications will become **heavy weight** and not suitable for small devices. | Modules follow **distributed structure** and applications will become **light** **weighted** and suitable for small devices. |
| Jar files approach **can’t be used for IOT devices** and **micro services**. | Module based approach **can be used for IOT devices** and **micro services**. |

## What is Jar Hell or Classpath Hell

The problem associated with the jar file is considered as Jar hell and as below,

1. NoClassDefFoundError in the middle of the program execution.
2. Version conflicts
3. Lack of security
4. Bigger size

## What are goals / benefits of JPMS

1. Reliable configuration
2. Strong encapsulation and security
3. Scalable java platform
4. Performance and memory improvements

## What is Module

A module is a group of packages similar to the jar file. But the specialty of module when compared to jar file is, module can contain configuration information inside a special file called as **module-info.java**.

## Module-Info Structure

**module** module\_name {

Here we have to define dependencies like,

1. What other modules required by this module

**requires** moduleA;

**requires** moduleB;

1. What packages exported by this module,

**exports** pack2;

**exports** pack3;

}

## Steps to develop module-based application

In modular programming, package is mandatory and if we missed to add it then compiler will throw an error.

Module Program Example,

module **ModuleA**

{

}

Package **pack1**;

public class **Test** {

public static void main (String[] args) {

SOP (First Module Program);

}

}

**Step to compile the Module**

**javac --module-source-path src -d out -m moduleA**

-d -> for package

out-> Generated class file placed in out folder

**Step to run the Module**

**java --module-path out -m moduleA/pack1.Test**

**To view the files inside folder in tree structure from CMD,**

Navigate to the folderand type **tree /f**

## Case Studies

1. If module-info.java is not available then the code won’t compile and we will get error. Hence, module-info.java is mandatory for every module.

javac --module-source-path src -d out -m ModuleA

**Error: module ModuleA not found in module source path**

1. Every class inside module should be part of some package, otherwise we will get compile time error saying unnamed package is not allowed in named modules.

// Package **pack1**;

public class **Test**

public static void main (String[] args) {

SOP (First Module Program);

}

}

**Error: unnamed package is not allowed in named modules**

1. The module name should not ends with digit (like module1). Otherwise, we will get warning at compile time.

javac --module-source-path src -d out -m Module1

**Warning: [module] module name component Module1 should avoid terminal digits**

## Various ways to compile Module

1. javac --module-source-path src -d out -m ModuleA
2. javac --module-source-path src -d out --module ModuleA
3. javac --module-source-path src -d out src/moduleA/module-info.java src/moduleA/pack1/Test.java
4. javac --module-source-path src -d out c:/users/desktop/src/moduleA/module-info.java c:/users/desktop/src/moduleA/pack1/Test.java

## Various ways to run Module

1. java --module-path out -m ModuleA/Pack1.Test
2. java --module-path out --module ModuleA/Pack1.Test
3. java --module-path out --add-modules ModuleA Pack1.Test

## Transitive Dependencies

There is a possibility to grant access of the modules, on which our current module depends, to the module that uses our current module. The ‘requires transitive’ keyword helps to achieve this. This means all the modules that are using our module will get the access to transitive dependency automatically.

Let us assume, Module A has mandatory dependent Module B and it should be given as below,

module ModuleA {

requires ModuleB;

}

Here, if the ModuleB should be available to the Module which requires ModuleA, or we can say the Module which uses ModuleA can also want to access ModuleB, in this case we should use the keyword called transitive. The declaration should be given as below,

module ModuleX {

requires ModuleA;

}

module ModuleA {

requires **transitive** ModuleB;

}

## Optional Dependencies

If dependent module should be available at compile time but optional at runtime, then such type of dependency is called as Optional Dependency. We can specify optional dependency by using **static** Keyword.

Let us assume, Module A has mandatory dependent Module B and it should be given as below,

module ModuleA {

requires ModuleB;

}

If in case, the Module A has optional dependent Module C and it should be written as below,

module ModuleA {

requires **static** ModuleC;

}

**Note:** In Compile time, both Module B and Module C is compulsory available for Module A but in runtime, the Module C is optional.

## Cyclic Dependencies

If ModuleA depends on ModuleB and ModuleB depends on ModuleA, such type of dependency is called as Cyclic Dependency. Cyclic Dependencies between the modules is not allowed in Java.

There may be chance of cyclic dependency between more than 2 modules as well,

ModuleA requires ModuleB

ModuleB requires ModuleC

ModuleC requires ModuleA

## Qualified Exports

Sometimes, a module can exports its package to specific module instead of every module. Then the specified module only can access the package. Such type of exports are called as Qualified Exports.

Exports <pack1> to <module B>, <module C>;

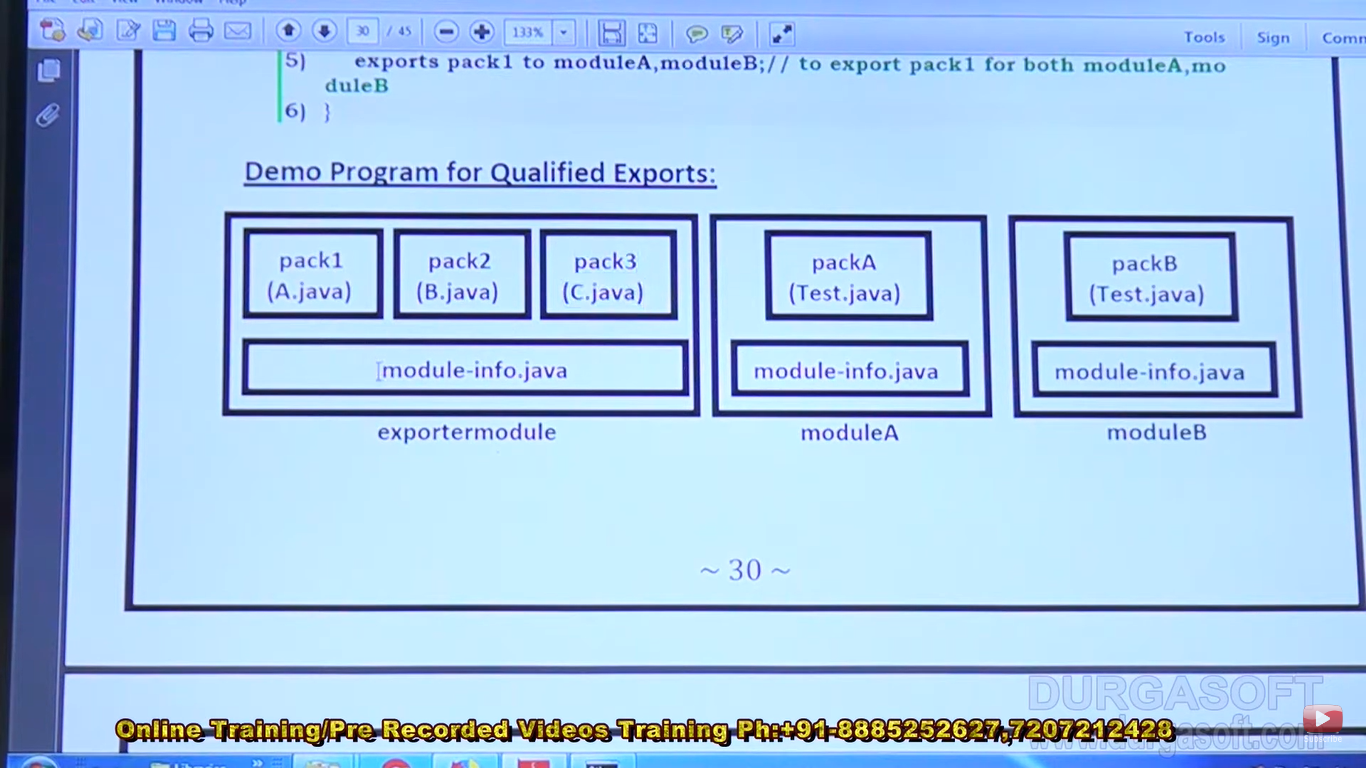
Example,

module ModuleA {

exports pack1; // To exports to all modules.

exports pack1 to ModuleB; // To export pack1 to ModuleB only.

exports pack1 to ModuleB, ModuleC; // To export pack1 to ModuleB & C only.



A.java

package pack1;

public class A {}

B.java

package pack2;

public class B {}

C.java

package pack3;

public class C {}

module-info.java

module exportModule {

exports pack1;

exports pack2 to moduleA;

exports pack3 to moduleA, moduleB;

}

**Module A**

Test.java

package packA;

import pack1.A;

import pack2.B;

import pack3.C;

public class Test {

public static void main (String[] args) {

SOP (“Qualified Exports Demo”);

}

}

module-info.java

module moduleA {

requires exportModule;

}

**Module B**

Test.java

package packB;

import pack1.A;

**import pack2.B; -> Compile Time Error**

import pack3.C;

public class Test {

public static void main (String[] args) {

SOP (“Qualified Exports Demo”);

}

}

## Module Graph

The dependencies between the modules can be represented by using a special graph, which is called as Module Graph.

Examples,

If moduleA requires moduleB then corresponding module graph is,

module moduleA {

requires moduleB;

}

Module A

Module B

If moduleB requires moduleC not only moduleB but also for the above modules of moduleB, then it can be written as below,

module moduleA {

requires moduleB;

}

module moduleB {

requires **transitive** moduleC;

}

**Notes:**

1. No cyclic dependencies between modules
2. No two modules should have same name
3. This module graph is used by Java Compiler and JVM

## Aggregator Module

Sometimes a group of modules can be reused by multiple other modules. Then it is not recommended to read each module individually. We can group those common modules into a single module, and we can read that module directly. This module which aggregates functionality of several modules into a single module is called as Aggregator module. If any module reads aggregator module, then automatically all its methods are by default available to that module.

Aggregator module won’t provide any functionality by its own, just it gathers and bundles together a bunch of other modules.

Aggregator module not required to contain a single java class. Just it requires transitive of all common modules.

If any module reads aggregate module automatically all the modules available in Aggregator are by default available to that module.

Example,

module **VisaModule** {

}

module **TravelModule** {

}

module **LodgeModule** {

}

module **GuideModule** {

}

module **TravelAgent** {

requires transitive VisaModule;

requires transitive TravelModule;

requires transitive LodgeModule;

requires transitive GuideModule;

}

module **Tourist** {

requires TravelAgent;

}

## Package Naming Conflicts

Two jar files can contain a package with same name, which may creates version conflicts and abnormal behaviour of the program at runtime.

But in Java9 module system, two modules can’t contain a package with same name. Otherwise, we will get compile time error. Hence in module system, there is no chance of version conflicts and abnormal behaviour of the program.

## Module Resolution Process (MRP) - Own Term by Tutor NOT given by JDK people

In the case of traditional classpath, JVM won’t check the required class files at the beginning. While executing program if JVM required any .class file, then only JVM will search in the classpath for the required .class file. If it is available then it will be loaded and used and if it is not available then at runtime we will get NoClassDefFoundError, which is not at all recommended.

But in module programming, JVM will search for the required modules in the module path before it starts execution. If any module is missing, in the beginning only JVM will identify and won’t start its execution. Hence, in module programming, there is no chance of getting NoClassDefFoundError in the middle of program execution.

To get the details of MRP while running the module program, we have to use the following option,

**java --module-path out --source-module-resolution -m useModule/Pack1.Test**

Example,

module **useModule** {

requires moduleA;

}

module **moduleA** {

requires moduleB;

}

module **moduleB** {

requires moduleC;

requires moduleD;

}

module **moduleC** {

}

module **moduleD** {

}

The module what we are trying to execute becomes the root module. Root module should contain the class with main method.

The main advantages of Module Resolution Process at beginning are,

1. We will get the error if any dependent module is not available.
2. We will get the error if multiple modules with the same name.
3. We will get the error if any cyclic dependency.
4. We will get the error if two modules contain packages with same name.

# Process API Updates (JEP-102)

Until Java 8, communication with processor/os/machine is very difficult. We required to write very complex native code and we have to use 3rd party jar files.

The way of communication with processor is varied from system to system. For example, In Windows, one way and in Mac other way. Being a programmer, we have to write code based on OS, which makes programming very complex.

To resolve this complexity, JDK 9 engineers introduced several enhancements to Process API. By using this updated API, we can write java code to communicate with any processor very easily. According to worldwide Java developers, Process API update is the number 1 feature in Java 9.

With this enhanced API, we can get the following items done,

1. Get the PID (Process ID) of the process
2. Create a new process
3. Destroy already running process
4. Get the process handles for processes
5. Get the complete information of process
6. Get the parent and child processes of running process
7. Get the process information like owner, children, etc.

Process Handle (Interface) is the newly introduced item as part of Java9.

## Methods to get Process Handle

To get the PH of current running process,

**ProcessHandle handle = ProcessHandle.current();**

To get the PH of given Process Object,

**ProcessHandle handle = p.toHandle();**

To get the PH of the specific PID

**Optional<ProcessHandle> obj = ProcessHandle.of(PID);**

**ProcessHandle handle = obj.get();**

## Use Case 1 - To get the PID of current running JVM.

public class **Test** {

public static void main (String[] args) throws Exception {

ProcessHandle handle = ProcessHandle.current();

long pid = handle.pid();

SOP (“The PID of JVM instance “+pid);

} }

## ProcessHandle.Info & Its Methods

We can get the complete information of a particular process by using ProcessHandle.Info object. We can get this Info object as follows,

ProcessHandle handle = ProcessHandle.current();

ProcessHandle.Info info = handle.info();

Once we got the Info object, we can call the following methods on that object.

### user ()

Return the user of the process.

Optional<String> obj = info.user();

SOP (obj.get());

### command ()

Returns the command that can be used to start the process.

Optional<String> obj = info.command();

SOP (obj.get());

### startInstant ()

Returns the start time of the process.

Optional<Instant> obj = info.startInstant();

SOP (obj.get());

### totalCPUDuration ()

Returns the total CPU time accumulated of the process.

Optional<Duration> obj = info.totalCPUDuration();

SOP (obj.get());

## Use Case 2a - To get the Complete Information of Current Running Process

public class **ProcessHandle\_Demo2** {

public static void main(String[] args) {

ProcessHandle handle = ProcessHandle.current();

ProcessHandle.Info processInfo = handle.info();

System.out.println("Complete Process Information => " + processInfo);

System.out.println("User => " + processInfo.user().get());

System.out.println("Command => " + processInfo.command().get());

System.out.println("StartTime => " + processInfo.startInstant().get());

System.out.println("Total CPU Time Acquired => " + processInfo.totalCpuDuration().get());

}

}

## Use Case 2b - To get the Complete Information of Already Exist and Running Process

public class **ProcessHandle\_Demo3** {

public static void main(String[] args) {

Optional<ProcessHandle> process = ProcessHandle.of(7364);

ProcessHandle handle = process.get();

ProcessHandle.Info processInfo = handle.info();

System.out.println("Complete Process Information => " + processInfo);

System.out.println("User => " + processInfo.user().get());

System.out.println("Command => " + processInfo.command().get());

System.out.println("StartTime => " + processInfo.startInstant().get());

System.out.println("Total CPU Time Acquired => " + processInfo.totalCpuDuration().get());

}

}

## ProcessBuilder

We can use ProcessBuilder to create Processes.

We can create ProcessBuilder object by using the following constructor.

**ProcessBuilder builder = new ProcessBuilder (String… args);**

Once we create a ProcessBuilder object, we can start the process by using start() method.

Examples,

ProcessBuilder pb = new ProcessBuilder(“notepad”, “names.txt”);

pb.start();

ProcessBuilder pb = new ProcessBuilder(“javac”, “Test.java”);

pb.start();

ProcessBuilder pb = new ProcessBuilder(“java”, “Test.java”);

pb.start();

## To Start and Destroy a Process from Java using ProcessBuilder

public class **ProcessBuilder\_Demo3** {

public static ProcessBuilder builder;

public static void main(String[] args) throws IOException, InterruptedException {

ProcessBuilder builder = new ProcessBuilder("java", "Test.java");

Process process = builder.start();

System.out.println("The process with PID " + process.pid() + " is started");

Thread.sleep(10000);

process.destroy();

System.out.println("The process with PID " + process.pid() + " is destroyed");

}

}

# JLINK

## Intro

Until Java 1.8v, to run a small java program (like hello world program) as well, we should use a bigger JRE which contains all java’s inbuilt 4300+ classes. It increases the size of Java Runtime Environment and Java applications. Due to this Java is not suitable for IOT devices and Micro Services.

To overcome this problem, Java people introduced Compact Profiles in Java 8. But they didn’t succeed that much. In Java 9, they introduced a permanent solution to reduce the size of Java Runtime Environment, which is nothing but JLINK.

JLINK is Java’s new command line tool (available in JDK\_HOME\bin) which allows us to link sets of only required modules (and their dependencies) to create a runtime image (our own JRE).

Our custom JRE contains only required modules and classes instead of all available classes. It reduces the size of JRE, which makes Java best suitable for IOT and microservices.

JLink also has a list of plugins (like compress) that will help optimize our solutions.

## Creation of our own JRE only with required modules

demoModule requires java.base.module. Hence add java.base module to out directory (Copy java.base.jmod from jdk-9\jmods to the out folder).

Out

|-java.base.jmod

|-demoModule

|-module-info.class

|-packA

|-Test.class

Now, we can create our own JRE with JLINK command.

C:\Users\Jaga\Desktop> jlink --module-path out --add-modules demoModule, java.base --output jagajre

We can run our application with our own custom JRE (jagajre) as follows,

C:\Users\Jaga\Desktop\jagajre\bin> java -m demoModule/packA.Test

**javac -d out --module-path out src/module-info.java**

**javac -d out --module-path out src/com/java9/jlink/module/demo/Test.java**

**java --module-path out -m Java9\_JLink\_Demo\_Module/com.java9.jlink.module.demo.Test**

**jdeps --module-path out -s -m Java9\_JLink\_Demo\_Module**

**jlink -v --module-path out --add-modules Java9\_JLink\_Demo\_Module --output customJRE**

## Compress Custom JRE

We can compress the size of the custom JRE with compress plugin.

Using --compress 0, 1, 2 we can compress the JRE.

## Provide own name to application with Launcher plugin

C:\Users\Jaga\Desktop> jlink --module-path out --add-modules demoModule, java.base --launcher demoapp=demoModule/packA.Test --compress 2 --output jagajre2

Now we can run our application only with the name demoapp.

C:\Users\Jaga\Desktop\jagajre2\bin>demoapp

If we set the path PATH=C:\Users\Jaga\Desktop\jagajre3\bin

Then we can run our application from anywhere.

**jlink -v --module-path out --add-modules Java9\_JLink\_Demo\_Module  --compress 2 --output compressedJRE**

**jlink -v --module-path out --add-modules Java9\_JLink\_Demo\_Module  --launcher demoapp= Java9\_JLink\_Demo\_Module/com.java9.jlink.module.demo.Test --compress 2 --output customJRE**

# HTTP/2 Client

## Intro

HTTP/2 client is one of the most exciting features, for which developers are waiting for long time. By using this new **HTTP/2** client, from java application we can send HTTP Request and we can process HTTP Response.

Prior to Java9, until java 1.8, we are using **HttpURLConnection** class to send HTTP Request and to process HTTP Response. It is the legacy class which was introduced as part of **JDK 1.1 (1997)**. There are several problems with this HttpURLConnection class.

## Problems with HttpURLConnection

1. Supports only HTTP/1.1 but not HTTP/2.
2. Only supports Text format but not binary data.
3. At a time only one request and network traffic would be high.
4. Supports only in blocking mode (Synchronous mode).
5. It is difficult to use.

Because of these problems, slowly developers started using 3rd party HTTP clients like Apache Http Client and Google Http Client, etc.

So, JDK engineers addresses these issues and introduced a brand-new HTTP/2 Client in Java 9.

## Advantages of HTTP/2 Client

1. Light weight and easy to use.
2. Supports HTTP/1.1 and HTTP/2. (Handles both Text data and Binary data).
3. Allows to send multiple requests at a time.
4. Supports both synchronous and Asynchronous mode.
5. It provides better performance and scalability when compared with HttpURLConnection.

## Important components of Java9 HTTP/2 Client

In Java9, HTTP/2 Client provided as Incubator module.

Module: jdk.incubator.httpclient

Package: jdk.incubator.http

Classes: HttpClient, HttpRequest and HttpResponse

**Note:** Incubator module is by default not available to our java application. Hence compulsory we should read explicitly by using requires directive.

module demoModule {

requires jdk.incubator.httpclient;

}

## Steps to send HttpRequest and Process HttpResponse from Java Application

1. Create HttpClient object
2. Create HttpRequest object
3. By using HttpClient object, send request and get HttpResponse
4. Process HttpRequest

**Create HttpClient object**

We can use HttpClient object to send HttpRequest to the Web server. We can create HttpClient object by using factory method. **newHttpClient()**

**HttpClient client = HttpClient.newHttpClient();**

**Creation of HttpRequest object**

We can create HttpRequest object as follows,

String url=”<http://www.google.com>”;

**HttpRequest req = HttpRequest.newBuilder(new URI(url)).GET().build();**

**Note**

newBuilder() method returns the builder object.

GET() method sets the request method of this builder to GET.

Build() method builds and returns a HttpRequest.

**By using HttpClient object, send request and get HttpResponse**

HttpClient contains the following methods,

1. send () to send synchronous request (blocking mode)
2. sendAsync () to send Asynchronous request (non-blocking mode)

HttpResponse resp = client.send(req, HttpResponse.BodyHandler.asString());

HttpResponse resp = client.send(req, HttpResponse.BodyHandler.asFile(Paths.get(“abc.html”)));

**Note**

BodyHandler is a functional interface present inside the HttpResponse. It can be used to handle body of HttpResponse.

**Process HttpRequest**

HttpResponse contains the status code, response headers and body.

HttpResponse class contains the following methods to retrieve data from the response.

statusCode()

body()

headers()

## Case Studies

class **Test** {

public static void main (String[] args) {

HttpClient client = HttpClient.newHttpClient();

String url = “<https://www.redbus.in/info/aboutus>”;

HttpRequest request = HttpRequest.newBuilder(new URI(url)).GET().build();

HttpResponse response = client.send(request, HttpResponse.BodyHandler.asString());

HttpResponse response = client.send(request, HttpResponse.BodyHandler.asFile(Paths.get(“abc.html”)));

SOP (“Status Code”+response.statusCode());

SOP (“Response Body”+response.body());

HttpHeaders header = response.headers();

Map<String, List<String>> map = header.map();

SOP (“Response Headers”);

map.forEach((k,v) -> SOP (K + “ : “ +v));

}

}

# JShell

## Intro

JShell is Java’s REPL (Read, Evaluate, Print, and Loop) tool. JShell is also known as interactive console.

By using this tool, we can execute java code snippets and we can get the 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.

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

## Limitations

1. Not designed for Main coding, we can use just to test small coding snippets, which can be used in our Main Coding.
2. Not replacement for IDEs like Eclipse, NetBeans, etc.
3. Not impressive feature. Other programming languages like Python, Swift, Lisp, Scala, and Ruby already have this REPL tool.

## Getting started with JShell

Start and stop JShell from the command prompt in verbose mode.

jshell -v **(To start the Jshell**)

jshell > /exit **(To stop the Jshell)**

**Note**

If any information displaying on the Jshell starts with ‘|’ is the information to the Programmer from the Jshell.

Jshell> 10+20

$1==> 30

| Created scratch variable $1: int

Jshell> SOP (“Java 9”)

Java 9

**Note**

Terminating semicolons are automatically added to the end of the complete snippet by Jshell if not entered.

## Default Packages Available in Jshell

We are not required to import java.lang.package. Because, by default its available. We can get the details of the available default packages in the Jshell using the **/imports** command.

## Jshell Compiler

Internally Jshell having a java compiler which is responsible to check syntax. If any violation, we can get Compile time error which is exactly same as normal compile time errors.

In our program, if there is any chance of getting checked exception, then we required to handle either by try-catch or by throws keyword. Otherwise, we will get Compile time error.

**PrintWriter pw = new PrintWriter(“abc.txt”);**

In normal java compiler, we will get compile time error saying **unreported exception FileNotFoundException;** must be caught or declared to be thrown.

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

## Conclusions

1. From the jshell we can execute any expressions and any java statements
2. Most of the packages are not required to import to the Jshell because by default already available to the Jshell.
3. Internally Jshell uses 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.

## Getting Help from JShell

To know list of options available with JShell

C:> jshell --help

To know the version of Jshell

C:> jshell --version

To know introduction of Jshell

Jshell > /help intro

To know the list of commands

Jshell > /help

To know the information about a particular command

Jshell > /help **commandName**

To get just names of all commands without description

Jshell > / **<Tab>**

To know the list of options available for a particular command

Jshell > /**<commandName> -<Tab>**

To get all the active snippets

Jshell > /list

## Understanding Jshell Snippet

Everything what allowed in Java is snippet. It can be Expression, Declarations, Statements, Classes, Interfaces, Methods, Variables, Import, etc. We can use all these as snippets. Only thing which is not allowed as snippet is package statement.

## Startup Snippets

To get the details of the startup snippets,

Jshell > /list -start

To get the details of all the active / error / startup snippets,

Jshell > /list -all

## Scratch Variable

The variables which are created by jshell to hold the intermediate values is called as **scratch variable**.

## To list the snippets by Id / Name

Jshell > /list 1

Jshell > /list m1()

Jshell > /list x

Jshell > /list x s

## To drop the snippet by Id / Name

Jshell > /drop 1

Jshell > /drop m1()

We can execute the snippets directly by using id with the command **/id**

Example,

Jshell > /1

## Working with Jshell Variables

There are 2 types of variables available in Jshell as below,

1. Explicit variable
2. Implicit (Scratch) variable

The variables which are created by the Programmer explicitly based on our requirement is called as Explicit Variables.

Example,

Jshell> int x = 10;

x ==> 10

| created variable x:int

Sometimes, Jshell itself create variables implicitly to hold temporary values, such type of variables are called as Implicit variables.

Jshell> 10+10;

$2 ==> 20

| created scratch variable $2: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 same name are not allowed.

Jshell > int x = 10;

Jshell > String x = “Java”;

x ==> Java

|replaced variable x: String

|update overwrote variable x: int

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

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

Jshell > String x = true;

|Error

|incompatible types: boolean cannot be converted to java.lang.String

**Note**

By using /vars command, we can list the variable’s type, name and value of all variables which are created by Jshell.

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

**To get the details of the active variable,**

Jshell> /vars

**To get the details of all variables,**

Jshell> /vars -all

**To drop the variable**

Jshell> /drop $4

**We can create complex variables in Jshell as well,**

Jshell > List<String> topics = List.of(“Java”, “Spring”, “Hibernate”);

Jshell > List<List<String>> list = List.of(topics);

## Difference b/w print() and printf()

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.

## Working with Jshell Methods

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

Example,

Jshell> public void m1 () {

SOP (“Hello”);

}

|Created method m1()

Jshell> m1()

Hello

In Jshell, there may be chance of having multiple methods with same name but different argument types and such methods are called as 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)

We can list all the methods information by using **/methods** command.

Jshell> /methods

| void m1()

| void m1(int)

/methods <name>

/methods <id>

/methods -start

/methods -all

**Note**

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 the return types are different) i.e., In Jshell at a time only one method with unique 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)

In Jshell, we can create more complex methods as well,

Jshell> public void sum (int… x)

{

int total = 0;

for (int x1:x) {

total = total+x1;

}

SOP (“The Sum:” + total);

}

Jshell> sum (20, 30)

The Sum: 50

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

Jshell> public void m1 () {

SOP (x);

}

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

Jshell> m1()

| attempt to call method m1() which cannot be invoked until variable x is declared

Jshell> public void m1 () {

m2();

}

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

Jshell> m1()

| attempt to call method m1() which cannot be invoked until method m2() is declared

## Drop Methods

We can drop methods by name with /drop command. If multiple methods with same name is available, then we should drop it by using the 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> /drop m3

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) {}

## Using External Editor with Jshell

It is every difficult to type lengthy code from Jshell. To overcome this problem, Jshell provides inbuilt editor.

**To open the editor in Jshell,**

Jshell> /edit

The above command will open the default editor window.

**To set the external editor as Jshell editor,**

Jshell> jshell> /set editor "C:\\Program Files\\Notepad++\\notepad++.exe";

| Editor set to: C:\Program Files\Notepad++\notepad++.exe ;

The external editor by default available for the current session and once the session is completed, then Jshell default editor only will show.

**To make the external editor as permanent editor in Jshell,**

Jshell> /set editor -retain

The above command will make the external editor as permanent editor in Jshell.

**If suppose, we need to switch back to default Jshell editor,**

Jshell> /set editor -default

## Working with Classes, Interfaces and Enums in Jshell

In Jshell, we can declare classes, interfaces, enum as well.

We can use /types command to list out the created types like classes, interfaces, and enum.

**Class**

jshell> /edit

| created class Student

jshell> /types

| class Student

jshell> /list Student

1 : public class Student{

String name;

int rollno;

Student(String name, int rollno){

this.name = name;

this.rollno = rollno;

}

public String getName(){

return name;

}

public int getRollno(){

return rollno;

}

}

**Interface**

jshell> /edit

| created interface Interim

jshell> /types

| interface Interim

jshell> /list Interim

5 : interface Interim

{

public static void display(){

System.out.println("Method from Interface");

}

}

jshell> Interim.display();

Method from Interface

**Enum**

jshell> /edit

| created enum FreshJuice

jshell> /types

| enum FreshJuice

jshell> /list FreshJuice

7 : public enum FreshJuice{

Apple("Sweet"), Grape("Bitter");

String taste;

FreshJuice(String taste){

this.taste = taste;

}

public String getTaste(){

return taste;

}

}

jshell> FreshJuice.Apple.taste;

$8 ==> "Sweet"

| created scratch variable $8 : String

jshell> FreshJuice.Grape.getTaste();

$9 ==> "Bitter"

| created scratch variable $9 : String

## Loading and Saving Snippets in Jshell

We can load snippets from the file and save snippets to the file.

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

mySnippets.jsh

String s = “Java”;

We can load all the snippets from the file into Jshell with /open command as below,

**Jshell> /open mySnippets.jsh**

Saving Jshell Snippet to the File

We can save Jshell snippets into the file with /save command.

**Jshell> /help save**

The above command gives the various options available with save command.

**Note** If the specified file is not available then this saves command itself will create the file and save the snippets.

Jshell> /save active.jsh

Jshell> /save -all all.jsh

Jshell> /save -start start.jsh

Jshell> /save -history history.jsh

**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 as below,

**Jshell> /save D://Java\_Sessions//snippets.jsh**

## How to reload Previous State (Session) into Jshell

The below command is used to restore the previous session snippets of jshell into the current session.

**Jshell> /reload -restore**

## How to reset Jshell State

If we want to reset the Jshell to default state, then below command can be used.

Jshell> /reset

## Using Jar Files in Jshell

It is very easy to use external jar files in the Jshell. We can add Jar files to the Jshell in 3 ways.

1. From Environment Variables -> Classpath
2. From Command Prompt
3. From Jshell itself

**From Command Prompt**

We have to open the Jshell with --class-path option.

C:\User> jshell -v --class-path C:\oracle\jars\ojdbc6.jar

**From Jshell Itself**

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

jshell> /env --class-path C:\oracle\jars\ojdbc6.jar

**Note:** Internally Jshell will use environment variable classpath if we are not setting classpath explicitly.

## Customize Jshell startup

By default, the following snippet will be executed at the time of Jshell startup.

Jshell> /list -start

We can customize these startup snippets based on our requirement. Assume our required startup snippets are available in mystartup.jsh

To provide the snippets available in mystartup.jsh, we have to open the jshell as follows,

**C:\User> jshell -v --startup mystartup.jsh**

**Note** If we want DEFAULT imports startup snippets along with our own snippets then we have to open the Jshell as follows,

**C:\User> jshell -v --startup DEFAULT mystartup.jsh**

jshell> /list

List our own snippets.

jshell> /list -start

List the startup imports.

**Note** To import all JAVASE packages (almost around 170+ packages) at the time of startup we have to open the Jshell as follows,

**C:\User> jshell -v --startup JAVASE**

jshell> /list -start

List the JAVASE imports.

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

**C:\User> jshell -v --startup JAVASE mystartup.jsh**

jshell> /list

List our own snippets.

jshell> /list -start

List the JAVASE imports.

## Need of Printing Option at Startup

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

**C:\User> jshell -v --startup PRINTING**

jshell> /list -start

List the overloaded print methods.

**Note** Whenever we are using the PRINTING shortcut, then DEFAULT imports won’t come. Hence to get the DEFAULT imports and PRINTING shortcut simultaneously, we have to open Jshell as follows.

**C:\User> jshell -v --startup DEFAULT PRINTING**

**Note** Various allowed options with --startup is,

1. DEFAULT
2. JAVASE
3. PRINTING

## Shortcuts and Autocompletion of Commands

### Variable Creation Shortcut

Just type the value on the Jshell and then press “Shift+Tab” followed by “v” then complete variable declaration code will be generated. We have to provide only the name of the variable.

jshell> “Java” //Just press shift+tab followed by v

jshell> String = “Java” // Here, we have to provide only the name of the variable

### Auto Import Shortcut

Just type the class or interface name on the Jshell and press “Shift+Tab” followed by “I”. Then we will get the 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

### To get all the static members of a Class

jshell> ClassName.<Tab>

jshell> String.<Tab>

valueOf(

copyValueOf(

…

### To get all the instance members of a Class

jshell> ClassObjectReference.<Tab>

jshell> String s = “Java”;

jshell> s.<Tab>

charAt(

length(

…

### To get the signature and documentation of a method

jshell> ClassName.MethodName(<Tab>

jshell> ClassObjectReference.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>

**Note** Even this <Tab> shortcut applicable for our own classes and methods as well.

jshell> public void m1(int…x){}

| created method m1(int…)

jshell>m1(<Tab>

m1(

jshell> m1(<Tab>

Signatures:

void m1(int…x)