1-06-2020

Java 9 new features with explanation :-

As we know that after coming Java 8 version, on 18th March 2014, many new features came which make java a very powerful programming language in comparison to other languages.

Java 9 came on Sept 21 st 2017. will many more enhanced features such as :-

1. JShell :-
2. JPMS (Java Platform Module System) :- Instead of Jar file, modularized (modules).
3. JLink (Java Linker) :- Wonderful feature to create our own customized JRE, Portable devices, microservices, IOT devices etc.
4. HTTP/2 Client- API
5. Process API Updates - Complete process information capturing makes programmers to make secured applications over the web programming.
6. Private Methods inside Interface →
7. try with Resources Enhancements.
8. Factory methods to create unmodifiable Collections
9. Stream API Enhancements
10. <> operator (Diamond Operator enhancement)
11. SafeVarargs Annotation
12. G1 GC (Garbage First Garbage Collection)

Now we explain it one by one :-

1. **JShell**

JShell concept is a tool for very beginners to make them interested to learn java language. Basic operators, datatypes can be demonstrated like python. It is playground to learn java for very beginners.

Developers also can test their coding snippets. We can view to check our API that we developed recently using JShell prompt. It can be opened by typing JShell at command prompt.

2. **JPMS** Java Platform Module System.

In earlier versions —

— All (Classes, interfaces, enum) placed inside the package.

many packages are placed in a jar file

Sometimes we face problems like — in case of jar files: —

① NoClassDefFoundError
② Version Conflicts
③ Security Problem
④ JDK and JRE having monolithic structure and Bigger Size.

These problems with jar files are known as

Jar Hell

From Java 1.9, Jar is no more, there only module exists, hence modular programming.
In every module — there is module-info.java

which is all about that module.

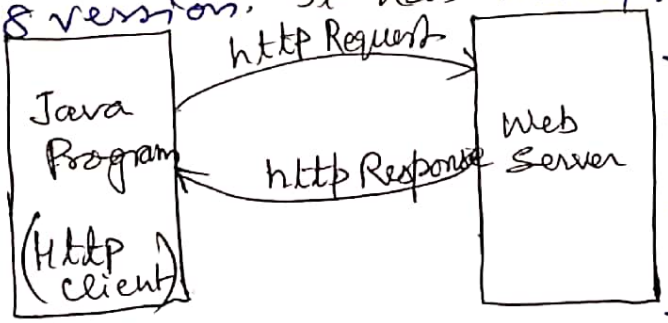
In modular programming, there is no run-time error about any class not found during execution. All problems are checked initially while compiling.

→ JPMS concept was started to develop in 2005. and finalized in 2017.

③ **JLink** Java Linker — JLink is the concept- came in Java 1.9 version. Java Runtime Environment- requires rt.jar files having 60MB size which contains more than 4300 classes in java 1.8. version and earlier. Even for a small program to compile and run, we have to load entire 60MB size rt.jar file, that requires more memory to run.

In Java 1.9, we ~~can~~ can create our own java Runtime Environment with the use of JLink concept- which requires lesser memory to run. (around 90% memory can be saved).

④ **Http/2** client API — In previous versions of Java we used class — HttpURLConnection — from 1997 onwards till Java 1.8 version. It has many limitations like :—



- Difficult to use
- At a time only one request.
- only text message can be sended.
- Blocking mode communication.

Due to this reason, many java developers started to use third party vendors — (http clients)

Apache Http Client-
Google Http Client-
} started to use for sending Http request to the web server

- After long waiting time, in java 9.0, new http Client came into picture in 2017.
  - **Http/2 client** can support Http/1 and Http/2 protocols versions.
- Now we are no longer need to use HttpURLConnection.

(5) **Process API Updates** — Communication with processes running inside the CPU was not easy in earlier versions of Java. In java 1.9 version, complete process information capturing is available for the programmers with safety features. The most successful development is Process API for Programmers.

Example:- Communicate with the processor:-

```
class Test {
public static void main(String[] args) throws Exception
{ long pid = ProcessHandle.current().pid();
  System.out.println("Current JVM Process ID:" + pid);
  Thread.sleep(100000);
} }
```

Explanation of above Program :-

ProcessHandle — Interface came in java 1.9 which is inside java.lang package, hence import is also not required.

current() — current running process

pid() — process ID No.

To see the output, we have to stop the running condition by sleeping the thread for a long interval (100 seconds), so that we can see the output of the program.

Compile & Run we can see the current PID Number and compare it with windows Task Manager's Process ID Number

In earlier version, Processor communication requires long programming code by using the native code API. Hence Java 1.9 is not only programmer friendly but Process friendly also.

(6) Private methods inside Interface —

Without affecting implementing classes, we can add new methods to an interface with adding default methods.

Interface functionality changes to versions to

Versions like :-

interface myinterface

Java Version 1.7

* { public methods
  public, static and final methods

+ Java version 1.8

{ default methods
  static methods

+ Java version 1.9

{ private methods

→ public, static, final, default, private methods

All features

```
interface int1 {
    m1();
    m2();
}  → default void m3(){ }

class T1 implements int1
{
    m1() { }
    m2() { }
// m3() not need to be implemented
}

class T2 implements int1
{
    m1() { }
    m2() { }
// m3() not need to implement.
}
```

If base interface int1 has to define a new methods then only default methods are introduced, that does not affect any of the implementing classes.

```
default void m3() { }
```

→ default methods need not to be implemented by implementing classes. This is the only way to add methods to an interface without affecting any of the implemented classes.

Private methods inside interfaces

In 1.7 Version: upto version 1.7, we can only place public methods inside an interface. Every variable in an interface is always public, static and final.

+

In Version 1.8: default methods, static methods

default methods — this is the only way without affecting implemented classes want to add methods.

```
default void m3()
{ }
```

+

In Version 1.9: private methods

Code-reusability purpose is achieved without affecting the implementation classes.

eg:-

```
interface Intef
{
    default void method1()
    {
        method2();
    }
    default void method2()
    {
        method2();
    }
    private void method2()
    {
        // common codes
    }
}
```

Code Reusability

(7) try with Resources Enhancements

try with Resources came in java 1.7 version

Syntax ~~is~~ is :-

```
try(R1 ; R2)
{
  ==
}
```

try with Resources Enhanced in java 1.9 now :-

In 1.8 ; Rule :-

1st way :-

```
try(FileWriter fw = new FileWriter(
  "abc.txt"); FileReader fr =
  new FileReader("input.txt"));
{
  ==
}
```

2nd way :-

```
try(FileWriter fw1 = fw;
    FileReader fr1 = fr)
{
  ==
}
```

In 1.9 ; Rule :-

```
FileWriter fw = new FileWriter("
    abc.txt");
FileReader fr = new FileReader(
    "input.txt");
try ( fw ; fr)
{
  ==
}
```

(Java 1.9 version)

(8) Factory methods to create unmodifiable (immutable) Collections :-

```
List<String> l = new ArrayList<String>();
l.add("A"); l.add("B"); l.add("C");
List<String> l2 = Collections.unmodifiableList(l);
```

In java early versions we have to write

the above 5 lines of codes to make unmodifiable Collection objects.

In Java 1.9 it is done by using a single line of code by Factory method as:—

```
List<String> l = List.of("A","B","C");
```

1 line

(9) Stream API Enhancements:—

Stream Concept came in Java 1.8 version — to process the objects from a collection.

Stream methods enhanced in version 1.9 are:—

takeWhile() } default methods
dropWhile() } of Stream API

Static methods { Stream.iterate() / Stream.ofNullable() } static methods introduced in java 1.9.

(10) <> operator:—

```
ArrayList<String> l = new ArrayList<String>();
```

— In 1.5 Vers

Enhanced in Java 1.7 Version:— diamond operator

```
ArrayList<String> l = new ArrayList<>();
```

Diamond operators are applicable for Normal classes only upto version 1.7.

But in Java 1.9 version, diamond operator is also applicable to anonymous inner classes also.

eg:-

```
ArrayList<String> l = new ArrayList<>()
{
    ____
};
```

(11) SafeVarargs Annotation

— This annotation came in Java 1.7. in which we can pass variable number of arguments in a function.

— If we use Varargs with Generics then there may be a chance of HeapPollution problem — Compiler will give warnings.

To overcome these problems — Java 1.7 Suppress these warnings created by the compiler.

Example:-

```
import java.util.*;
public class Test{
public static void main(String[] args){
List<String> l1 = Arrays.asList("A","B");
List<String> l2 = Arrays.asList("C","D");
m1(l1,l2);
```

```
    }
public static void m1(List<String>...l)
 {
   for(List<String> l1 : l)
   {
      System.out.println(l1);
   }
 }
}
```

Compile this code :- This will give warnings.

To Suppress compiler warnings we have to modify-

```
import java.util.*;
public class Test{
public static void main(String []args){
 List<String> l1 = Arrays.asList("A","B");
 List<String> l2 = Arrays.asList("C","D");
    m1(l1,l2);
  }
   @SafeVarargs
 public static void m1(List<String>...l)
  {  for(List<String> l1 :l)
     {
       System.out.println(l1);
     }
  }
}
```

until Java 1.8 version — we used SafeVarargs Annotations for —

- static methods
- final methods
- constructors only

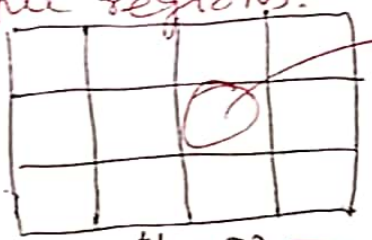
But for Java 1.9 version +

- private methods also can use

(12) G1GC — Garbage First Garbage Collector

We have several Garbage Collectors are there in Java

1. Serial Garbage Collector (1.1)
2. Parallel Garbage Collector (1.4 default) } based on Generations
3. Concurrent Mark Undersweep Garbage Collector (CMS) (1.4 added)
4. G1GC in 1.6 — Total Heap is divided into multiple regions.



Performance wise Best Garbage Collector introduced in java 1.7 known as G1GC. From Java 1.9 — onwards G1GC is the default Garbage Collector.

By default until Java 1.8, Parallel GC was the default garbage Collector used.

Factory Method for creating unmodifiable Collections :—

Factory method returns same class object on which it is defined — static Factory Methods (Java 1.9 version)

eg:—

① `Runtime rt = Runtime.getRuntime();`

② `DateFormat df = DateFormat.getInstance();`

③ `List l = List.of();`

④ `Set s = Set.of();`

⑤ `Map m = Map.of();`

⑥ `Map m = Map.ofEntries();`

//variable args method of() can take 0, 1, 2, ... 10

Now in JDK 1.9 no. of arguments are variable.

`var ... args` } 12 methods

Variable argument method

we can pass any no. of arguments.

In case of Map entries — 10 key-value pairs are directly passed with of() method.

When more than 10 key-value pairs are to be passed then we have to use ofEntries().

We can see :—

`javap java.util.Map` ↵

① When by miske we tried to modify any unmodifiable Collection object then — UnsupportedOperationException occurs

② Null value is not allowed with factory methods for creating unmodifiable collections — gives NullPointerException

③ Set and Map — if duplicate values are there — IllegalArgumentException can happen.

Until Java 1.8 version

```
ArrayList<String> l = new ArrayList<String>();
l.add("AP");
l.add("TS");
l.add("UP");
l.add("BR");
```

(modifiable) general Collection objects which can be modified, added, deleted.

In Java 1.9 onwards:—

```
List<String> l = List.of("AP", "TS", "UP", "BR");
```

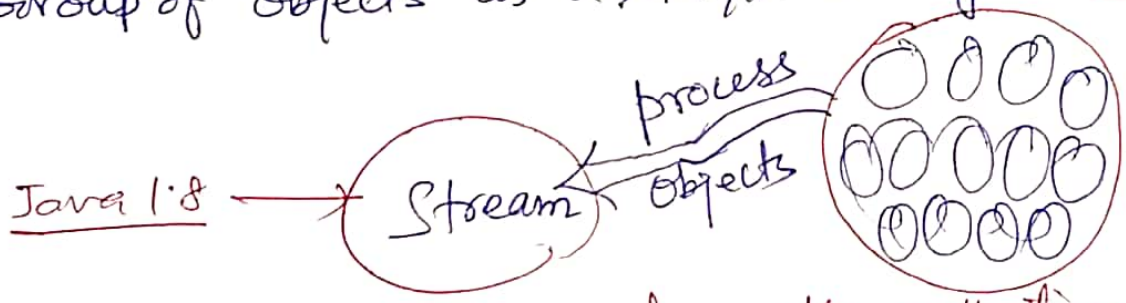
It is unmodifiable (immutable) collection objects which can not be modified later.

Stream API Enhancements :—

In Java 1.8 Version :—

Stream concept can be used for Collection.

Group of objects as a single entity — Collection



To process objects from the collections —

Stream → Filtering

java.io streams	VS	java.util Streams
① Related to stream of data from the File.		① Related to stream of Objects from the Collection.
② To process data from the File.		② To process Objects from the Collection.

Stream s = c.stream();

(Any Collection object.)

In Java 1.8 inside interface Collection

default Stream stream()
{
}.

Java 1.8

This interface is present in java.util.stream package defined

Now, we will learn to process Stream objects

There are three important methods — from Java 1.8 version

① filter()
② map()
③ flatMap()

① filter() — To filter objects of Stream based on any condition.

② map() — For every object, it is going to generate equivalent object by applying some function. eg: $x \longrightarrow x * x$ one-to-one

③ flatMap() :- For every object you may generate zero, one or more objects based on your requirements. One-to-Any

① filter(Predicate<T> t) → returns public Stream

Condition → Lambda Expression

```
ArrayList<String> l1 = new ArrayList<String>();
for(int i=0; i<=10; i++)
{
  l1.add(i);
}
```

```
ArrayList<Integer> l1 = new ArrayList<Integer>();
for(int i=0; i<=10; i++)
{
    l1.add(i);
}
System.out.print(l1); // [0,1,2,3,...,10]
```

in Java 1.7v without Stream :—

```
ArrayList<Integer> l2 = new ArrayList<Integer>();
for(Integer I : l1)
{
    if(I%2==0)
    {
        l2.add(I);
    }
}
System.out.print(l2); //[0,2,4,6,8,10]
```

Lambda Expression

In Java 1.8v with Stream :—

```
List<Integer> l2 = l1.stream().filter(I -> I%2==0).
                   collect(Collectors.toList());
System.out.println(l2); // [0,2,4,6,8,10]
```

Here, l1.stream() — for using Stream concept of java.util package.

filter(I → I%2==0) — filter upon a given condition, known as Lambda Expression.

Example:-

```
import java.util.*;
import java.util.stream.*;
public class Test{
    public static void main(String[]args)
    {
```

```
ArrayList<Integer> l1 = new ArrayList<Integer>();
for(int i=0; i<=10; i++)
    {
      l1.add(i);
    }
System.out.println(l1);

                    //output [0,1,2,....10]
List<Integer> l2 = l1.stream().filter(x -> x%2==0).
                        collect(Collectors.toList());
System.out.println(l2);
                                //output [0,2,4,6,8,10]
  }
}
```

In Java 1.9 enhancement of Stream API :-

Four methods added :- to enhance Stream API

① takeWhile() } default methods added in java 1.9 version
② dropWhile() }

③ Stream.iterate() } Static methods added in java 1.9 version
④ Stream.ofNullable() }

① Syntax :-

takeWhile(Predicate)

It is boolean condition

To take elements while a Predicate condition is true. As soon as condition is false - exit (stop).

In filter method - every element is traversed (processed) but takeWhile() doesnot guarantees - because as condition becomes false - exits from the list (Collection).

```
ArrayList <Integer> l1 = new ArrayList<Integer>();
l1.add(2); l1.add(4); l1.add(1); l1.add(5);
l1.add(6); l1.add(8);
System.out.println(l1); // [2, 4, 1, 5, 6, 8] printed.
List <Integer> l2 = l1.stream().filter(x -> x%2==0).collect(
                        Collectors.toList());
System.out.println("After filter() :" + l2); // [2, 4, 6, 8] printed
List <Integer> l3 = l1.stream().takeWhile(x -> x%2==0).collect(
                        Collectors.toList());
System.out.println("After takeWhile() :" + l3); // [2,4] printed.
```

Do: Let us try to execute them in a program.

Now we try to learn about JPMS in detail :-

JPMS — In place of rt.jar as present in previous versions of java, new java has module system — which contains group of packages.

Some module names are —

java.base , java.sql , java.rmi , java.logging , java.desktop ..... etc.

java.base :—

java.sql :—

java.rmi

java.logging :—

java.desktop :—

(awt & swing)

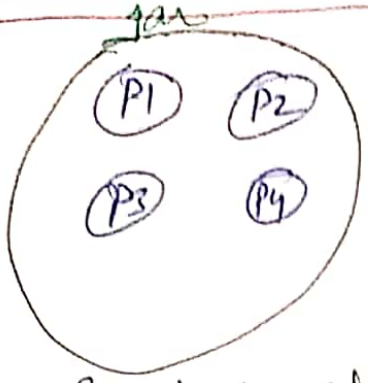
(98 modules are present)

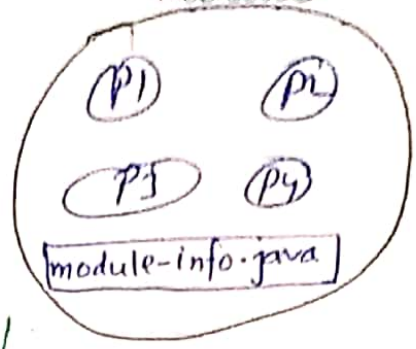
```
System.out.println(String.class.getModule());
System.out.println(Connection.class.getModule());
System.out.println(ArrayList.class.getModule());
```

Each module has a special file regarding module configuration —

module-info.java

Difference between Jar file & Java Module:






```
module moduleA
{
requires moduleB;
}
```

Error of missing classes at run-time → NoClassDefFound

No run-time error about missing any class.

(1) Jar is a 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.

(2) 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.

(3) In the classpath the order of jar files important and JVM will always considers from left to right for the required .class files. If multiple jars contain the same class then there may be a chance of Version conflicts and results abnormal behavior of our 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.

class-path	module-path
Security Problems	No Security Problems
Monolithic Structure (heavy weight)-application	Distributed structure (light weight)-application

Small portable devices, microservices, IOT devices programs can be easily developed by Module based approach language such as Java 9.

Q What are various dis-advantages with jar files

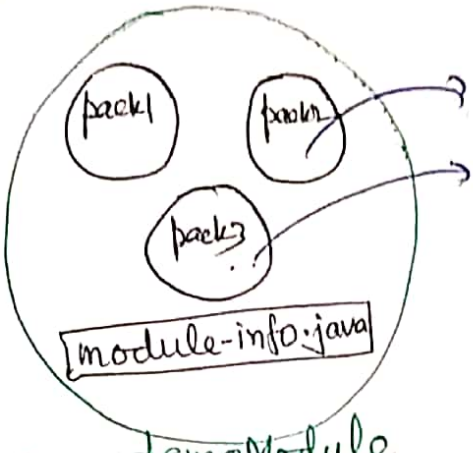
① NOClassDefFoundError in middle of program execution
② Version conflicts
③ Lack of Security
④ Bigger Size

All these problems are combinedly known as – **Jar Hell**

Q What are various goals of JPMS concept ?

① Reliable configuration – Runtime error-NO chance.
② Strong Encapsulation and Security — Only exported packages can be accessible from outside of module
③ Scalable java platform — within the small memory we could be able to run java application.
④ Performance and Memory improvements.

Module →



demoModule

Suppose, We have lō make pack1 for private use only for internal module classes only.

module-info.java

```
module module_name
{
    Here we have lō define module dependencies like
① What other modules required by this module
    requires moduleA;
    requires moduleB;
② What packages exported by this module
        exports pack2;
        exports pack3;

    ---
    ---

}
```