***Java Platform Module System (JPMS) Introduction:***

Modularity concept introduced in Java 9 as the part of Jigsaw project. It is the main important concept in java 9. The development of modularity concept started in 2005.

The First JEP(JDK Enhancement Proposal) for Modularity released in 2005. Java people tried to release Modularity concept in Java 7(2011) & Java 8(2014).

But they failed. Finally after several postponements this concept introduced in Java 9.

Until Java 1.8 version we can develop applications by writing several classes, interfaces and enums.

We can places these components inside packages and we can convert these packages into jar files.

By placing these jar files in the classpath, we can run our applications. An enterprise application can contain 1000s of jar files also.

Hence jar file is nothing but a group of packages and each package contains several .class files.

But in Java 9, a new construct got introduced which is nothing but 'Module'. From java 9 version onwards we can develop applications by using module concept.

Module is nothing but a group of packages similar to jar file. But the specialty of module when compared with jar file is, module can contain configuration information also.

Hence module is more powerful than jar file. The configuration information of module should be specified in a special file named with module-info.java Every module should compulsory contains module-info.java, otherwise JVM won't consider that as a module of Java 9 platform.

What is the need of JPMS? Application development by using jar file concept has several serious problems. Problem-1: Unexpected NoClassDefFoundError in middle of program execution There is no way to specify jar file dependencies until java 1.8V.At runtime,if any dependent jar file is missing then in the middle of execution of our program, we will get NoClassDefFoundError, which is not at all recommended.

***What is the need of JPMS?***

Application development by using jar file concept has several serious problems.

**Problem-1:** Unexpected NoClassDefFoundError in middle of program execution There is no way to specify jar file dependencies until java 1.8V.At runtime,if any dependent jar file is missing then in the middle of execution of our program, we will get NoClassDefFoundError, which is not at all recommended.

But in Java9, there is a way to specify all dependent modules information in module-info.java. If any module is missing then at the beginning only, JVM will identify and won't start its execution. Hence there is no chance of raising NoClassDefFoundError in the middle of execution.

**Problem 2:** Version Conflicts or Shadowing Problems If JVM required any .class file, then it always searches in the classpath from left to right until required match found.

But in java9 module system, there is a way to specify dependent modules information for every module seperately.JVM will always consider only required module and there is no order importance. Hence version conflicts won't be raised in Java 9.

**Problem 3: Security problem**

There is no mechanism to hide packages of jar file. Assume pack1 can be used by other jar files, but pack2 is just for internal purpose only. Until Java 8 there is no way to specify this information.

Everything in jar file is public and available to everyone. Hence there may be a chance of Security problems. public is too much public in jar files.

But in Java 9 Module system, we can export particular package of a module.

Only this exported package can be used by other modules. The remaining packages of that module are not visible to outside.

Hence Strong encapsulation is available in Java 9 and there is no chance of security problems. Even though class is public, if module won't export the corresponding package, then it cannot be accessed by other modules. Hence public is not really that much public in Java 9 Module System

**Problem 4:** JDK/JRE having Monolithic Structure and Very Large Size The number of classes in Java is increasing very rapidly from version to version.

JDK 1.0V having 250+ classes

JDK 1.1V having 500+ classes ...

JDK 1.8V having 4000+ classes

And all these classes are available in rt.jar.

Hence the size of rt.jar is increasing from version to version.

The size of rt.jar in Java 1.8Version is around 60 MB. To run small program also, total rt.jar should be loaded, which makes our application heavy weight and not suitable for IOT applications and micro services which are targeted for portable devices

It will create memory and performance problems also. (This is something like inviting a Big Elephant in our Small House: Installing a Heavy Weight Java application in a small portable device).

But in java 9, rt.jar removed. Instead of rt.jar all classes are maintained in the form of modules. Hence from Java 9 onwards JDK itself modularized. Whenever we are executing a program only required modules will be loaded instead of loading all modules, which makes our application light weighted.

Now we can use java applications for small devices also. From Java 9 version onwards, by using JLINK , we can create our own very small custom JREs with only required module

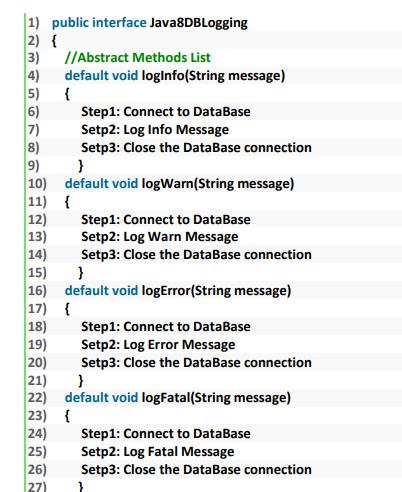
**2) Private Methods in Interfaces**

**Need of private Methods inside interface**:

If several default methods having same common functionality then there may be a chance of

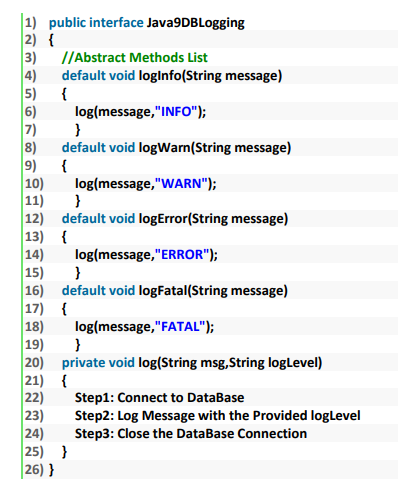
duplicate code (Redundant Code).

In the above code all log methods having some common code,which increases length of the code and reduces readability.It creates maintenance problems also. In Java8 there is no solution for this.



**How to declare private Methods inside interface:**

JDK 9 Engineers addresses this issue and provided private methods inside interfaces. We can seperate that common code into a private method and we can call that private method from every default method which required that functionality.



**Advantages of private Methods inside interfaces:**

The main advantages of private methods inside interfaces are:

1. Code Reusability

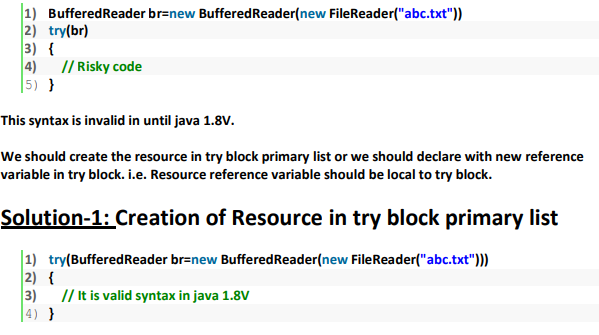
2. We can expose only intended methods to the API clients (Implementation classes), because interface private methods are not visible to the implementation classes.

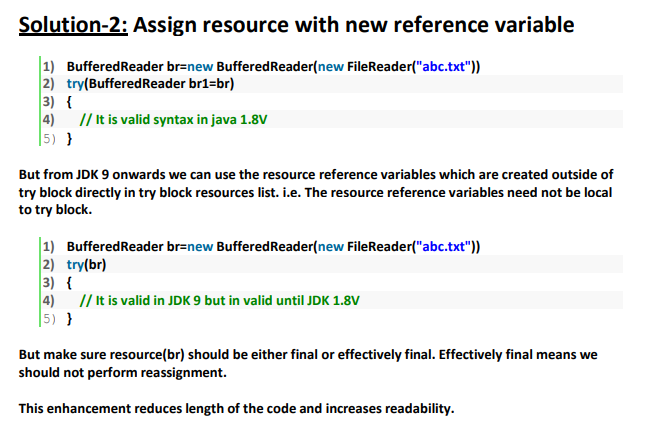
**Try with Resources Enhancements**

**Problems with JDK 7 Try with Resources:**

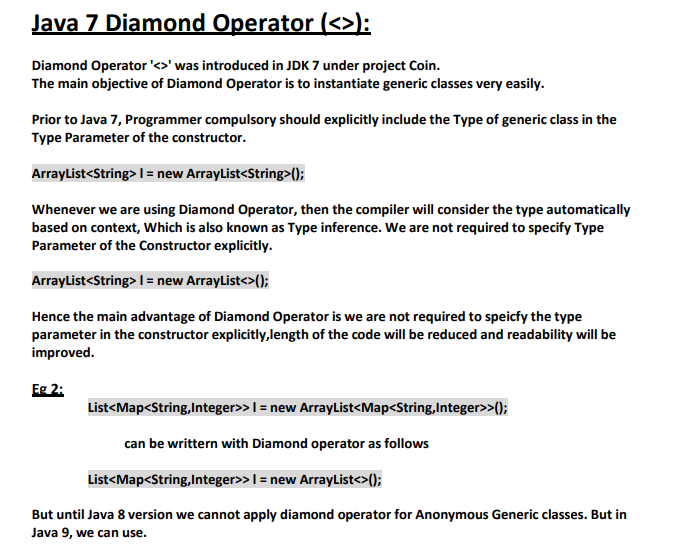
1. The resource reference variables which are created outside of try block cannot be used directly

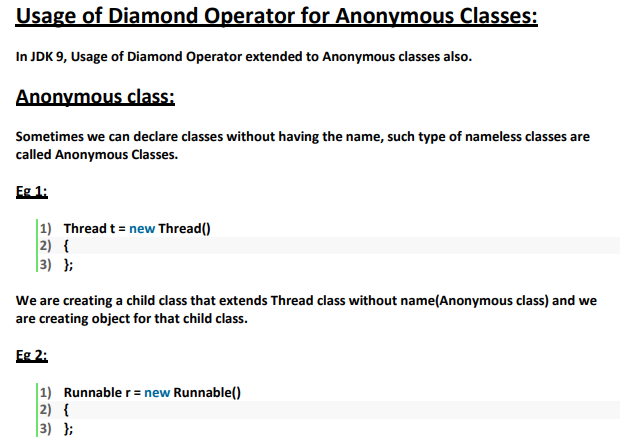
in try with resources.

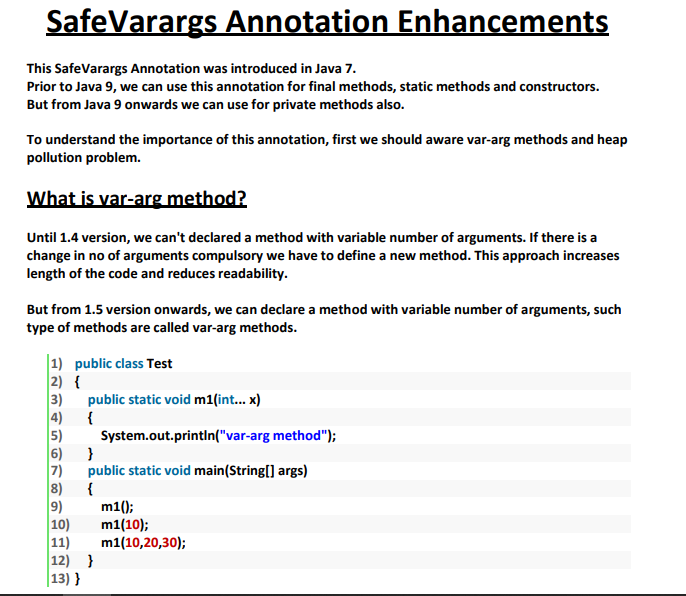




***Diamond Operator Enhancements***

******

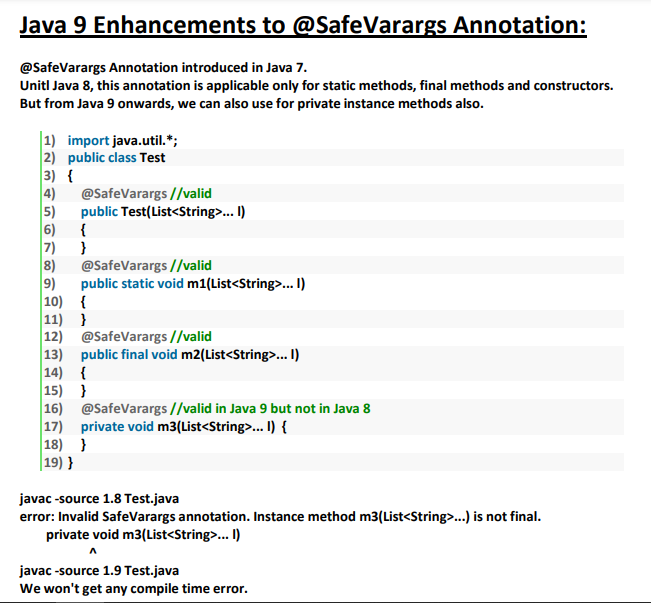
****

****

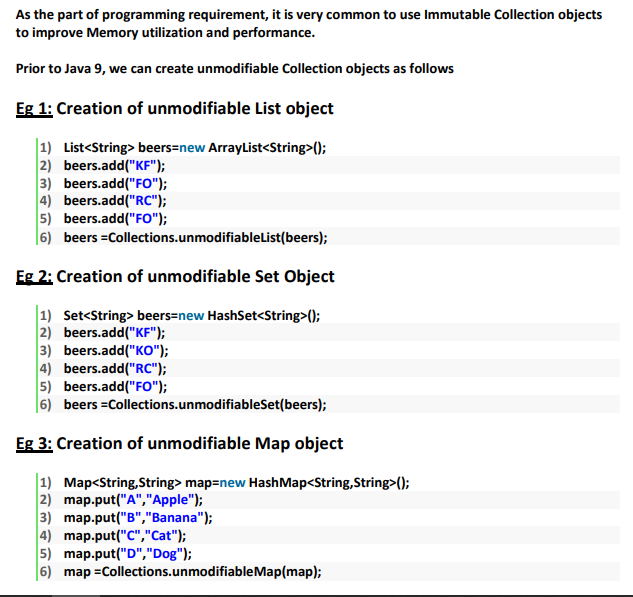
**Output**

**var-arg method**

**var-arg method**

**var-arg method**

**Factory Methods for creating unmodifiable Collections**

****

**Eg: To create unmodifiable List with Java 9 Factory Methods.**

List<String> beers = List.of("KF","KO","RC","FO");

It is very simple and straight forward way.

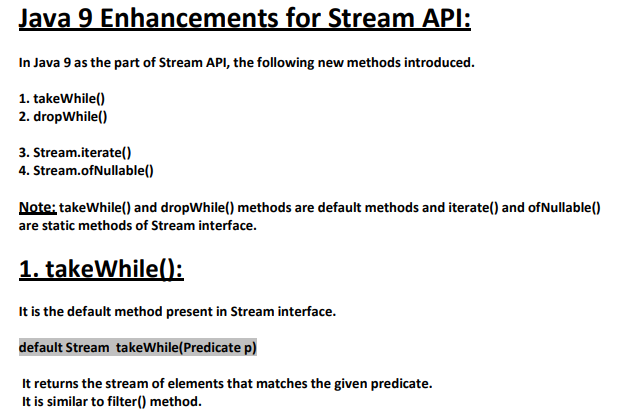
**Note:**

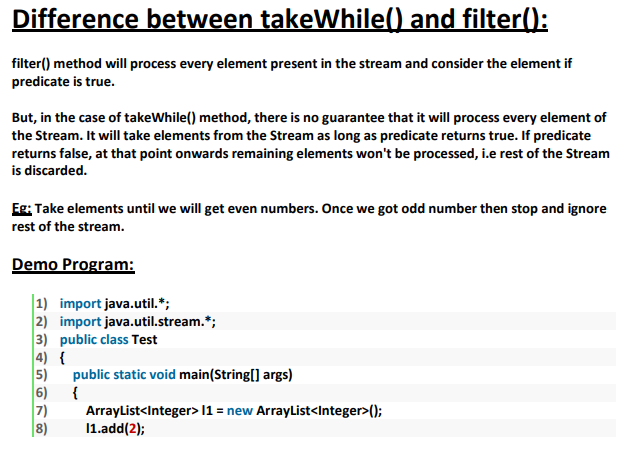
**1**. While using these factory methods if any element is null then we will get NullPointerException.

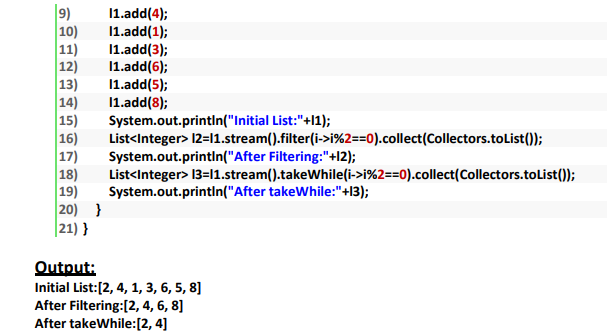
**List<String> fruits = List.of("Apple","Banana",null);=>NullPointerException**

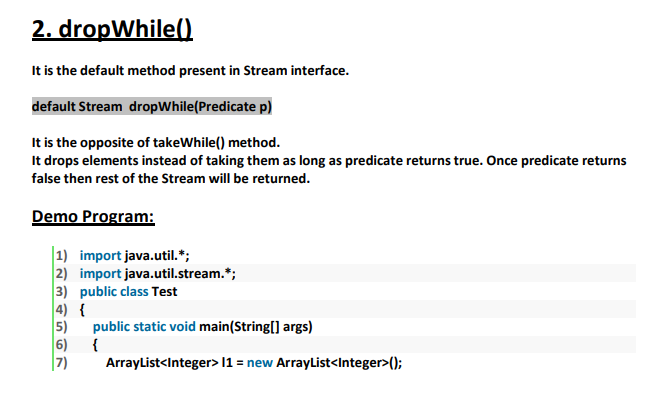
**2.** After creating the List object,if we are trying to change the content(add|remove|replace elements)then we will get **UnsupportedOperationException** because List is immutable(unmodifiable)

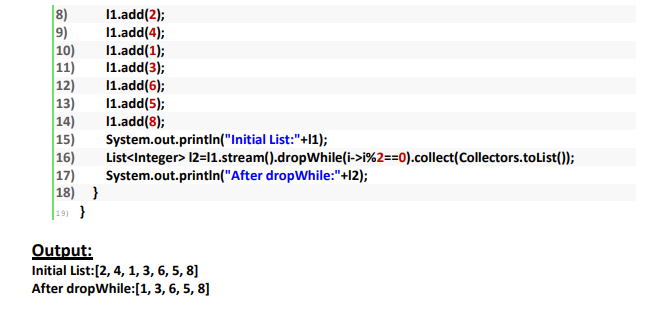
***Stream API Enhancements***

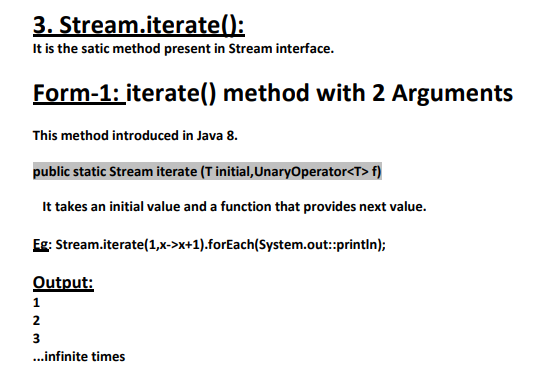












**How to limit the number of iterations:**

For this we can use limit() method.

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

Output:

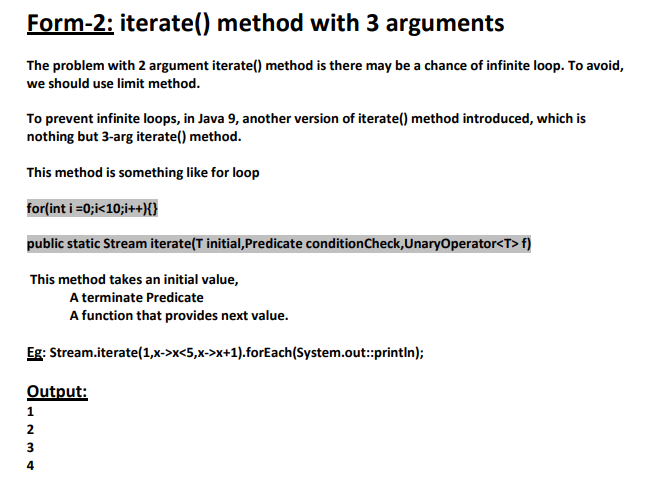
1

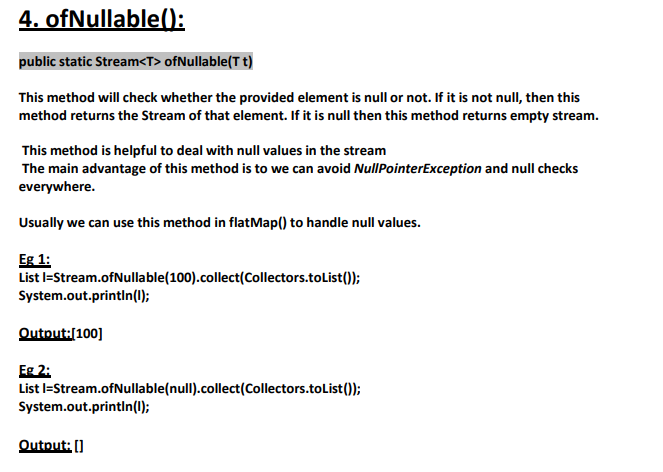
2

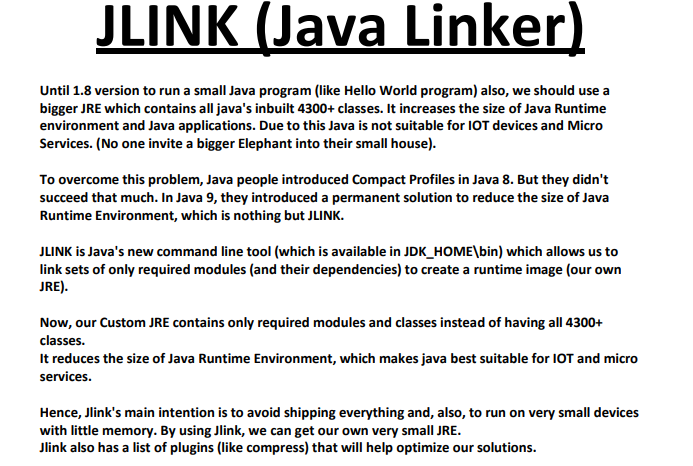
3

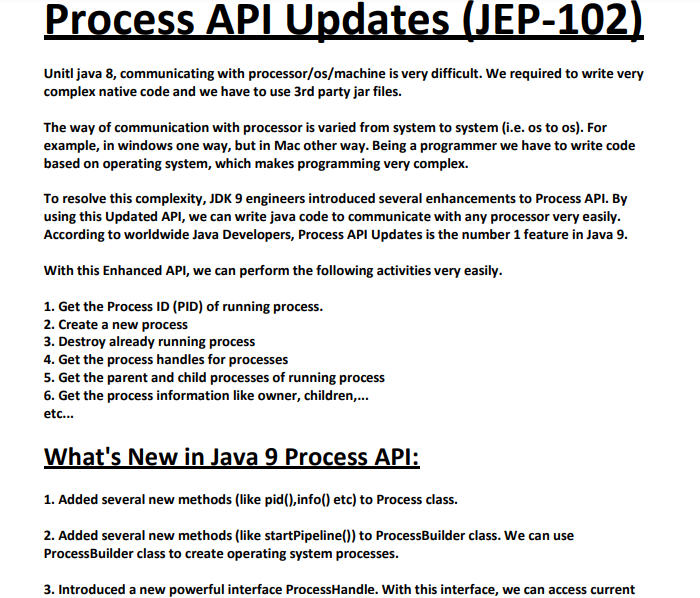
4

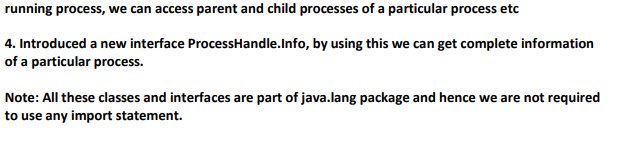
5

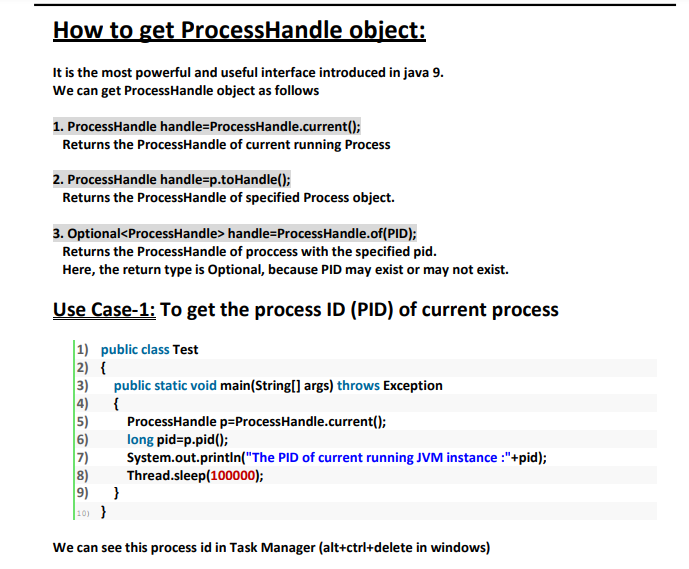


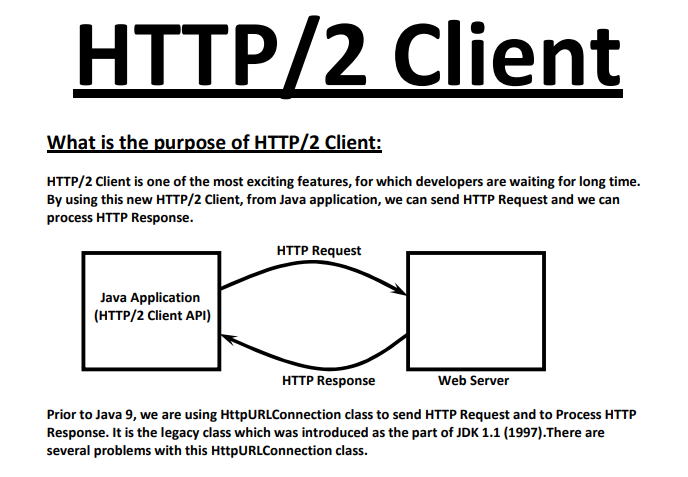












**Problems with Traditional HttpURLConnection class:**

1. It is very difficult to use.

2. It supports only HTTP/1.1 protocol but not HTTP/2(2015) where

A. We can send only one request at a time per TCP Connection, which creates network traffic

problems and performance problems.

B. It supports only Text data but not binary data

3. It works only in Blocking Mode (Synchronous Mode), which creates performance problems.

Because of these problems, slowly developers started using 3rd party Http Clients like Apache Http

client and Google Http client etc.

Advantages of Java 9 HTTP/2 Client:

1. It is Lightweight and very easy to use.

2. It supports both HTTP/1.1 and HTTP/2.

3. It supports both Text data and Binary Data (Streams)

4. It can work in both Blocking and Non-Blocking Modes (Synchronous Communication and

Asynchronous Communication)

5. It provides better performance and Scalability when compared with traditional

HttpURLConnection.

etc...

JDK 9 Engineers addresses these issues and introduced a brand new HTTP/2 Client in Java 9.