Java 9

1.. Private Methods Inside Interface

If several default methods having same common functionality, then there may be a chance of duplicate code (Redundant Code).

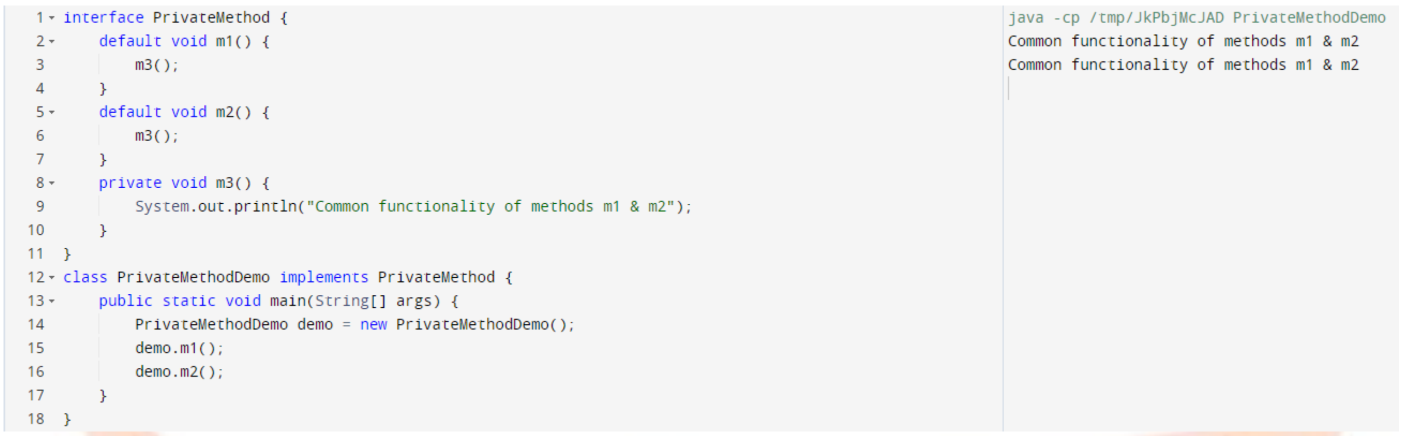
```java
interface DLogging {
    default void logInfo(String message) {
        //1. Connect to DB
        //2. Log Info Message
        //3. Close DB Connection
    }
    default void logError(String message) {
        //1. Connect to DB
        //2. Log Error Message
        //3. Close DB Connection
    }
    default void logWarning(String message) {
        //1. Connect to DB
        //2. Log Warning Message
        //3. Close DB Connection
    }
}
```

In the above code all log methods having some common code like Connect to Database and Closing Database Connection, which increases length of the code and reduces readability. It creates maintenance problems also. In Java8 there is no solution for this. In **Java 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.

```java
interface DLogging {
    default void logInfo(String message) {
        log(message,"INFO");
    }
    default void logError(String message) {
        log(message,"ERROR");
    }
    default void logWarning(String message) {
        log(message,"WARNING");
    }
    private void log(String message, String logLevel) {
        //1. Connect to DB
        //2. Log Message with the Provided Log Level
        //3. Close DB Connection
    }
}
```

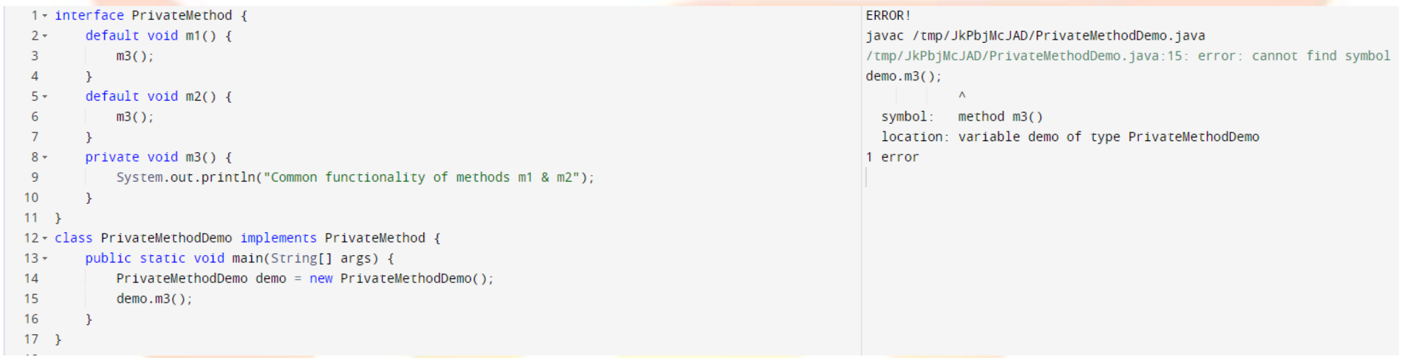
Private instance methods will provide code reusability for default methods.

Example Snippet



```java
interface PrivateMethod {
    default void m1() {
        m3();
    }
    default void m2() {
        m3();
    }
    private void m3() {
        System.out.println("Common functionality of methods m1 & m2");
    }
}
class PrivateMethodDemo implements PrivateMethod {
    public static void main(String[] args) {
        PrivateMethodDemo demo = new PrivateMethodDemo();
        demo.m1();
        demo.m2();
    }
}
```

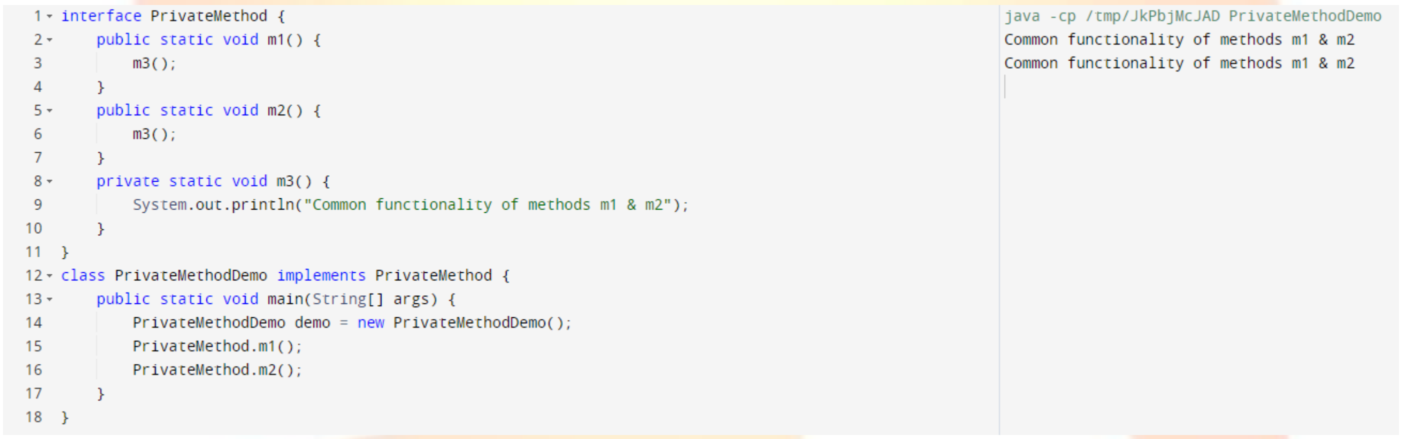
```
java -cp /tmp/JkPbjMcJAD PrivateMethodDemo
Common functionality of methods m1 & m2
Common functionality of methods m1 & m2
```



```java
interface PrivateMethod {
    default void m1() {
        m3();
    }
    default void m2() {
        m3();
    }
    private void m3() {
        System.out.println("Common functionality of methods m1 & m2");
    }
}
class PrivateMethodDemo implements PrivateMethod {
    public static void main(String[] args) {
        PrivateMethodDemo demo = new PrivateMethodDemo();
        demo.m3();
    }
}
```

```
ERROR!
javac /tmp/JkPbjMcJAD/PrivateMethodDemo.java
/tmp/JkPbjMcJAD/PrivateMethodDemo.java:15: error: cannot find symbol
demo.m3();
    ^
  symbol:   method m3()
  location: variable demo of type PrivateMethodDemo
1 error
```

Inside Java 8 interfaces, we can take public static methods also. If several static methods having some common functionality, we can seperate that common functionality into a private static method and we can call that private static method from public static methods where ever it is required.



```java
interface PrivateMethod {
    public static void m1() {
        m3();
    }
    public static void m2() {
        m3();
    }
    private static void m3() {
        System.out.println("Common functionality of methods m1 & m2");
    }
}
class PrivateMethodDemo implements PrivateMethod {
    public static void main(String[] args) {
        PrivateMethodDemo demo = new PrivateMethodDemo();
        PrivateMethod.m1();
        PrivateMethod.m2();
    }
}
```

```
java -cp /tmp/JkPbjMcJAD PrivateMethodDemo
Common functionality of methods m1 & m2
Common functionality of methods m1 & m2
```

Note: Interface static methods should be called by using interface name only even in implementation classes also.

Advantages of private Methods inside interfaces:

The main advantages of private methods inside interfaces are:

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

Note:

- private methods cannot be abstract and hence compulsory private methods should have the body.
- private method inside interface can be either static or non-static.

2.. try with resources enhancements

There was a problem when Try with Resources feature was launched in Java 7. The problem is that the resource reference variables which are created outside of try block cannot be used directly in try with resources.

```java
class TryWithResourceDemo {
    public static void main(String[] args) {
        BufferedReader br=new BufferedReader(new FileReader("abc.txt"));
        try(br) {
            //Risky Code
        }
    }
}
```

This syntax is invalid in until java 1.8V. But we have a workaround for this. Either We should create the resource in try block primary list or we should declare with new reference variable in try block. i.e. Resource reference variable should be local to try block.

```java
class TryWithResourceDemo {
    public static void main(String[] args) {
        try(BufferedReader br=new BufferedReader(new FileReader("abc.txt"))) {
            //Risky Code
        }
    }
}
```

```java
class TryWithResourceDemo {
    public static void main(String[] args) {
        BufferedReader br=new BufferedReader(new FileReader("abc.txt"));
        try(BufferedReader br1 = br) {
            //Risky Code
        }
    }
}
```

But from Java 9 onwards we can use the resource reference variables which are created outside of try block directly in try block resources list. i.e. The resource reference variables need not be local to try block. But make sure resource should be either final or effectively final. Effectively final means we should not perform reassignment. This enhancement reduces length of the code and increases readability.

```java
class TryWithResourceDemo {
    public static void main(String[] args) {
        BufferedReader br=new BufferedReader(new FileReader("abc.txt"));
        try(br) {
            //Risky Code
            //It is Valid in Java 9
        }
    }
}
```

3.. Diamond Operator Enhancements

This enhancement is as the part of Milling Project Coin (JEP 213). Diamond Operator '<>' was introduced in JDK 7 under project Coin. The main objective of Diamond Operator is to instantiate generic classes very easily. Prior to Java 7, Programmer compulsory should explicitly include the Type of generic class in the

Type Parameter of the constructor.

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

Whenever we are using Diamond Operator, then the compiler will consider the type automatically based on context, Which is also known as **Type inference**. We are not required to specify Type

Parameter of the Constructor explicitly.

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

Hence the main advantage of Diamond Operator is 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.

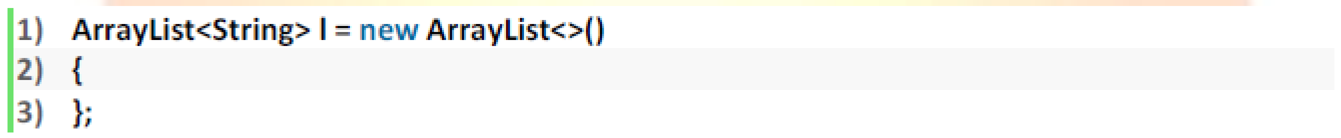
Example

List<Map<String, Integer>> l = new ArrayList<Map<String, Integer>>();

can be written with Diamond operator as follows

List<Map<String, Integer>> l = new ArrayList<>();

But until Java 8 version we cannot apply diamond operator for Anonymous Generic classes. But in Java 9 Usage of Diamond Operator extended to Anonymous classes also.



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

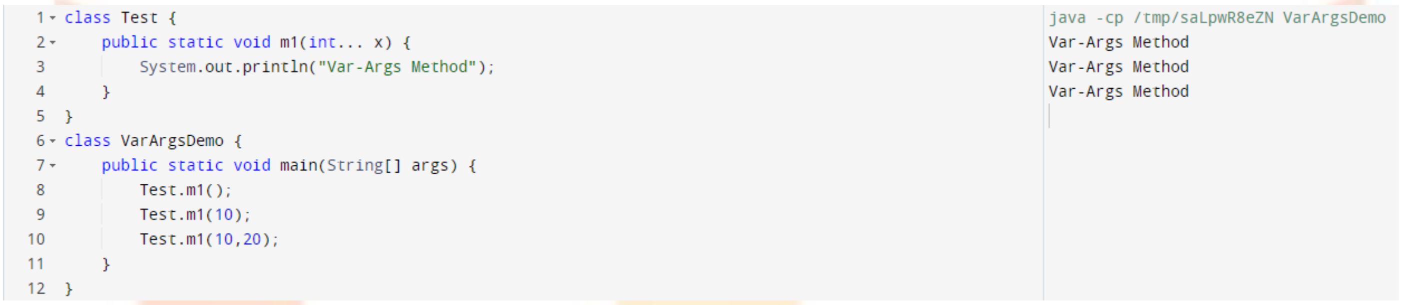
It is valid in Java 9 but invalid in Java 8.

4.. SafeVarargs Annotation Enhancements

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

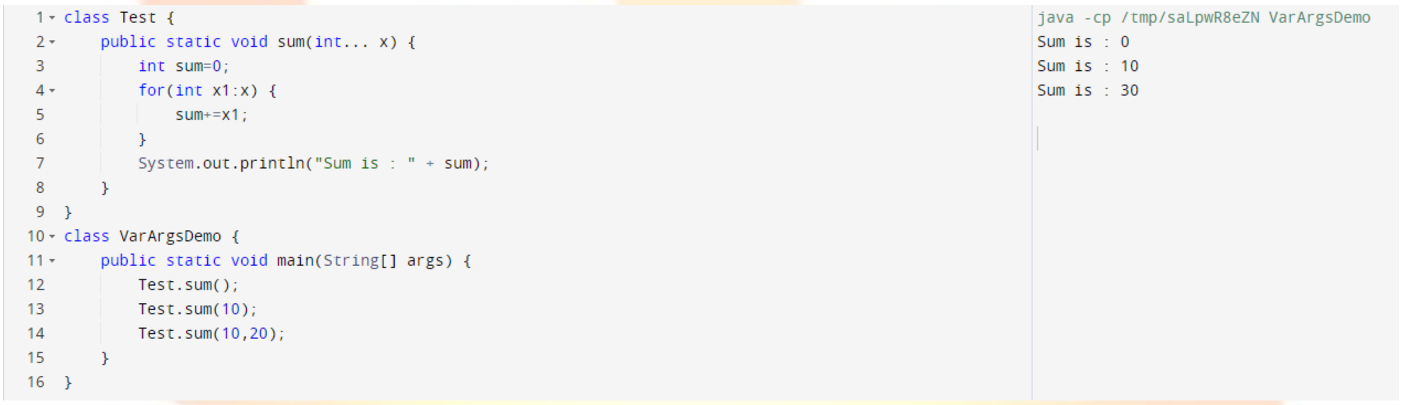
What is var-arg method?

Until 1.4 version, we can't declare a method with variable number of arguments. If there is a change in no of arguments compulsory, we have to define a new method. This approach increases length of the code and reduces readability. But from 1.5 version onwards, we can declare a method with variable number of arguments, such type of methods is called **var-arg methods**.



```java
class Test {
    public static void m1(int... x) {
        System.out.println("Var-Args Method");
    }
}
class VarArgsDemo {
    public static void main(String[] args) {
        Test.m1();
        Test.m1(10);
        Test.m1(10,20);
    }
}
```

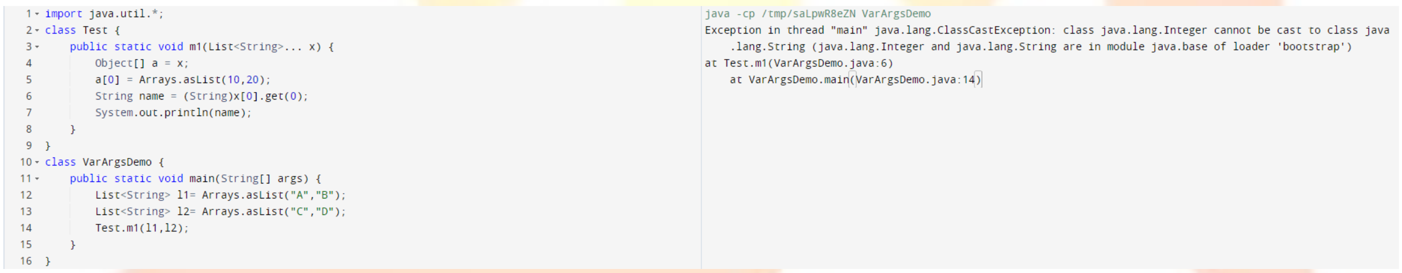
```
java -cp /tmp/saLpwR8eZN VarArgsDemo
Var-Args Method
Var-Args Method
Var-Args Method
```



```java
class Test {
    public static void sum(int... x) {
        int sum=0;
        for(int x1:x) {
            sum+=x1;
        }
        System.out.println("Sum is : " + sum);
    }
}
class VarArgsDemo {
    public static void main(String[] args) {
        Test.sum();
        Test.sum(10);
        Test.sum(10,20);
    }
}
```

```
java -cp /tmp/saLpwR8eZN VarArgsDemo
Sum is : 0
Sum is : 10
Sum is : 30
```

If we use var-arg methods with Generic Type then there 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 **ClasssCastException**. This problem is called **Heap Pollution**. In our code, if there is any chance of heap pollution then compiler will generate warnings.



```java
import java.util.*;
class Test {
    public static void m1(List<String>... x) {
        Object[] a = x;
        a[0] = Arrays.asList(10,20);
        String name = (String)x[0].get(0);
        System.out.println(name);
    }
}
class VarArgsDemo {
    public static void main(String[] args) {
        List<String> l1= Arrays.asList("A","B");
        List<String> l2= Arrays.asList("C","D");
        Test.m1(l1,l2);
    }
}
```

```
java -cp /tmp/saLpwR8eZN VarArgsDemo
Exception in thread "main" java.lang.ClassCastException: class java.lang.Integer cannot be cast to class java
    .lang.String (java.lang.Integer and java.lang.String are in module java.base of loader 'bootstrap')
at Test.m1(VarArgsDemo.java:6)
    at VarArgsDemo.main(VarArgsDemo.java:14)
```

While compiling above program we will get some warnings.

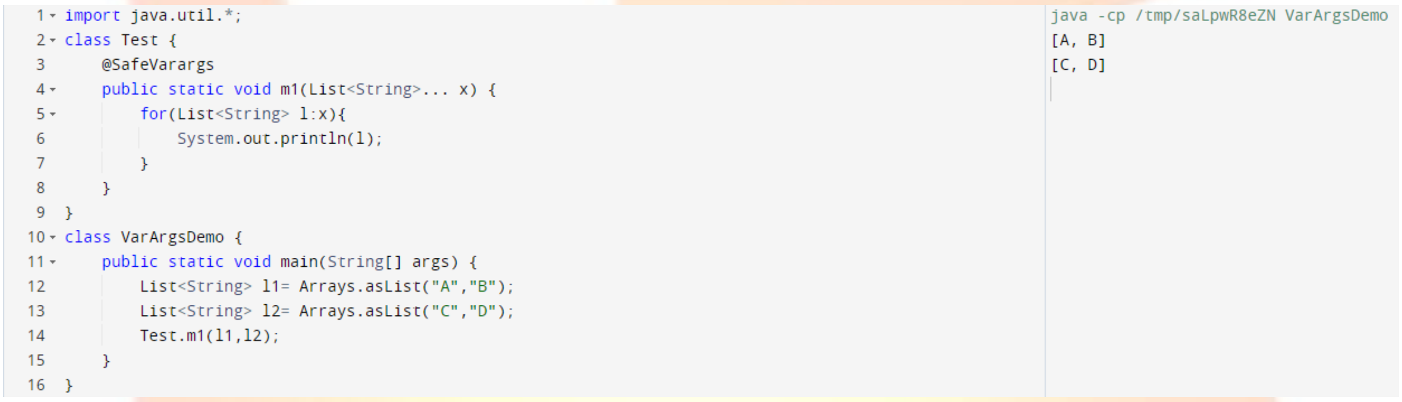
Note: Test.java uses unchecked or unsafe operations.

Note: 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(l1,l2);
             ^
warning: [unchecked] Possible heap pollution from parameterized vararg type List<String>
      public static void m1(List<String>... l)
                             ^
2 warnings
```

Need of @SafeVarargs Annotation:

Very few Var-arg Methods cause **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.



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

```
java -cp /tmp/saLpwR8eZN VarArgsDemo
[A, B]
[C, D]
```

In the program, inside m1() method we are not performing any reassignments. Hence there is no chance of Heap Pollution Problem. Hence, we can suppress Compiler generated warnings with **@SafeVarargs** annotation.

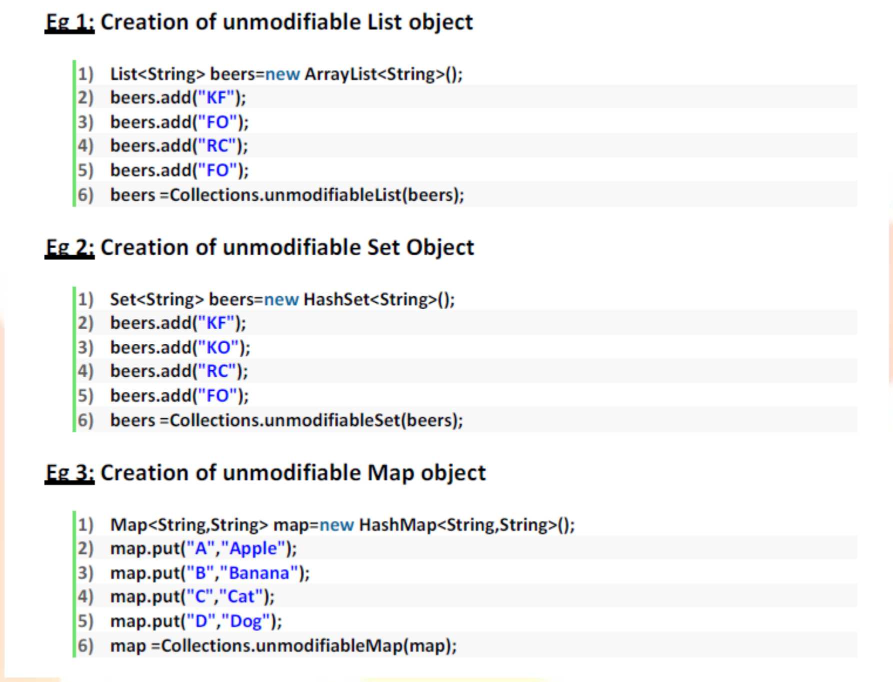
Note: At compile time observe the difference with and without **SafeVarargs** Annotation.

This SafeVarargs Annotation was introduced in Java 7. Prior to Java 9, we can use this annotation for final methods, static methods and constructors. But from Java 9 onwards we can use for private methods also.

```java
import java.util.*;
public class Test
{
   @SafeVarargs //valid
   public Test(List<String>... l)
   {
   }
   @SafeVarargs //valid
   public static void m1(List<String>... l)
   {
   }
   @SafeVarargs //valid
   public final void m2(List<String>... l)
   {
   }
   @SafeVarargs //valid in Java 9 but not in Java 8
   private void m3(List<String>... l) {
   }
}
```

5.. Factory Methods to Create Unmodifiable Collections

As the part of programming requirement, it is very common to use Immutable Collection objects to improve Memory utilization and performance. Prior to Java 9, we can create unmodifiable Collection objects as follows.



Eg 1: Creation of unmodifiable List object

```
List<String> beers=new ArrayList<String>();
beers.add("KF");
beers.add("FO");
beers.add("RC");
beers.add("FO");
beers =Collections.unmodifiableList(beers);
```

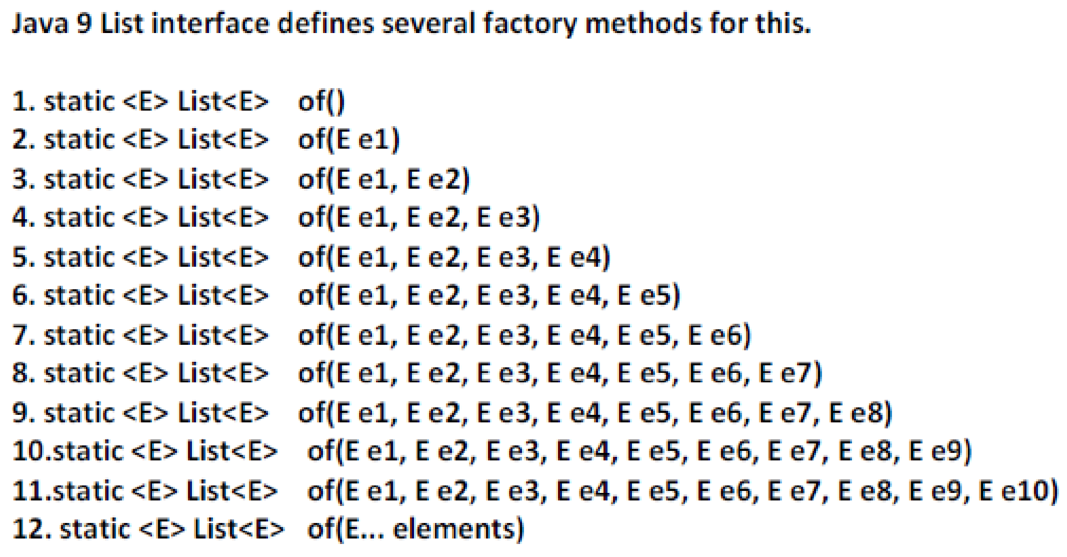
Eg 2: Creation of unmodifiable Set Object

```
Set<String> beers=new HashSet<String>();
beers.add("KF");
beers.add("KO");
beers.add("RC");
beers.add("FO");
beers =Collections.unmodifiableSet(beers);
```

Eg 3: Creation of unmodifiable Map object

```
Map<String,String> map=new HashMap<String,String>();
map.put("A","Apple");
map.put("B","Banana");
map.put("C","Cat");
map.put("D","Dog");
map =Collections.unmodifiableMap(map);
```

This way of creating unmodifiable Collections is verbose and not convenient. It increases length of the code and reduces readability. JDK Engineers addresses this problem and introduced several factory methods for creating unmodifiable collections.



Java 9 List interface defines several factory methods for this.

```
1. static <E> List<E>   of()
2. static <E> List<E>   of(E e1)
3. static <E> List<E>   of(E e1, E e2)
4. static <E> List<E>   of(E e1, E e2, E e3)
5. static <E> List<E>   of(E e1, E e2, E e3, E e4)
6. static <E> List<E>   of(E e1, E e2, E e3, E e4, E e5)
7. static <E> List<E>   of(E e1, E e2, E e3, E e4, E e5, E e6)
8. static <E> List<E>   of(E e1, E e2, E e3, E e4, E e5, E e6, E e7)
9. static <E> List<E>   of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8)
10.static <E> List<E>   of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9)
11.static <E> List<E>   of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)
12. static <E> List<E>  of(E... elements)
```

Up to 10 elements the matched method will be executed and for more than 10 elements internally var-arg method will be called. JDK Engineers identified List of up to 10 elements is the common requirement and hence they provided the corresponding methods. For remaining cases **var-arg** method will be executed, which is very costly. These many methods just to improve performance.

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.

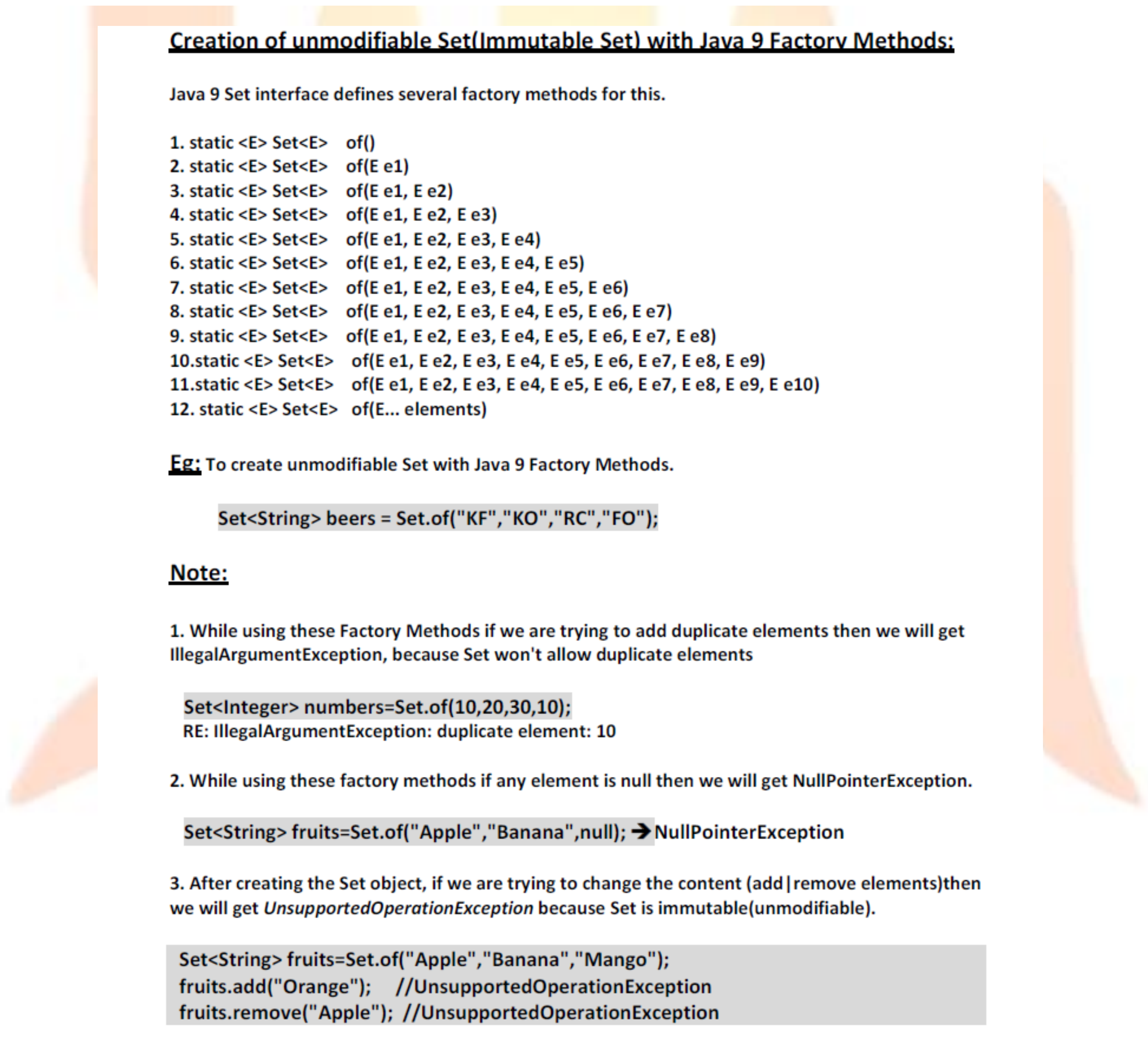
Note:

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).

```
List<String> fruits=List.of("Apple","Banana","Mango");
fruits.add("Orange");    //UnsupportedOperationException
fruits.remove(1);       //UnsupportedOperationException
fruits.set(1,"Orange"); //UnsupportedOperationException
```



Creation of unmodifiable Set(Immutable Set) with Java 9 Factory Methods:

Java 9 Set interface defines several factory methods for this.

1. static <E> Set<E>   of()
2. static <E> Set<E>   of(E e1)
3. static <E> Set<E>   of(E e1, E e2)
4. static <E> Set<E>   of(E e1, E e2, E e3)
5. static <E> Set<E>   of(E e1, E e2, E e3, E e4)
6. static <E> Set<E>   of(E e1, E e2, E e3, E e4, E e5)
7. static <E> Set<E>   of(E e1, E e2, E e3, E e4, E e5, E e6)
8. static <E> Set<E>   of(E e1, E e2, E e3, E e4, E e5, E e6, E e7)
9. static <E> Set<E>   of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8)
10. static <E> Set<E>   of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9)
11. static <E> Set<E>   of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)
12. static <E> Set<E>   of(E... elements)

Eg: To create unmodifiable Set with Java 9 Factory Methods.

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

Note:

1. While using these Factory Methods if we are trying to add duplicate elements then we will get IllegalArgumentException, because Set won't allow duplicate elements

```
Set<Integer> numbers=Set.of(10,20,30,10);
RE: IllegalArgumentException: duplicate element: 10
```

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

```
Set<String> fruits=Set.of("Apple","Banana",null); →NullPointerException
```

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

```
Set<String> fruits=Set.of("Apple","Banana","Mango");
fruits.add("Orange");    //UnsupportedOperationException
fruits.remove("Apple");  //UnsupportedOperationException
```

Creation of unmodifiable Map (Immutable Map) with Java 9 Factory Methods:

Java 9 Map interface defines of() and ofEntries() Factory methods for this purpose.

1. static <K,V> Map<K,V>   of()
2. static <K,V> Map<K,V>   of(K k1,V v1)
3. static <K,V> Map<K,V>   of(K k1,V v1,K k2,V v2)
4. static <K,V> Map<K,V>   of(K k1,V v1,K k2,V v2,K k3,V v3)
5. static <K,V> Map<K,V>   of(K k1,V v1,K k2,V v2,K k3,V v3,K k4,V v4)
6. static <K,V> Map<K,V>   of(K k1,V v1,K k2,V v2,K k3,V v3,K k4,V v4,K k5,V v5)
7. static <K,V> Map<K,V>   of(K k1,V v1,K k2,V v2,K k3,V v3,K k4,V v4,K k5,V v5,K k6,V v6)
8. static <K,V> Map<K,V>   of(K k1,V v1,K k2,V v2,K k3,V v3,K k4,V v4,K k5,V v5,K k6,V v6,K k7,V v7)

9. static <K,V> Map<K,V>   of(K k1,V v1,K k2,V v2,K k3,V v3,K k4,V v4,K k5,V v5,K k6,V v6,K k7,V v7,K k8,V v8)
10. static <K,V> Map<K,V>   of(K k1,V v1,K k2,V v2,K k3,V v3,K k4,V v4,K k5,V v5,K k6,V v6,K k7,V v7,K k8,V v8,K k9,V v9)
11. static <K,V> Map<K,V>   of(K k1,V v1,K k2,V v2,K k3,V v3,K k4,V v4,K k5,V v5,K k6,V v6,K k7,V v7,K k8,V v8,K k9,V v9,K k10,V v10)
12. static <K,V> Map<K,V>   ofEntries(Map.Entry<? extends K,? extends V>... entries)

Note:

Up to 10 entries,it is recommended to use of() methods and for more than 10 items we should use ofEntries() method.

Eg: `Map<String,String> map=Map.of("A","Apple","B","Banana","C","Cat","D","Dog");`

How to use Map.ofEntries() method:

Map interface contains static Method entry() to create immutable Entry objects.

```
Map.Entry<String,String> e=Map.entry("A","Apple");
```

This Entry object is immutable and we cannot modify its content. If we are trying to change we will get RE: UnsupportedOperationException

`e.setValue("Durga");` ➔ UnsupportedOperationException

By using these Entry objects we can create unmodifiable Map object with Map.ofEntries() method.

Note:

1. While using these Factory Methods if we are trying to add duplicate keys then we will get IllegalArgumentException: duplicate key.But values can be duplicated.

```
Map<String,String> map=Map.of("A","Apple","A","Banana","C","Cat","D","Dog");
RE:  java.lang.IllegalArgumentException: duplicate key: A
```

2. While using these factory methods if any element is null (either key or value) then we will get NullPointerException.

```
Map<String,String> map=Map.of("A",null,"B","Banana"); ==>NullPointerException
Map<String,String> map=Map.ofEntries(entry(null,"Apple"),entry("B","Banana"));
                    ➔ NullPointerException
```

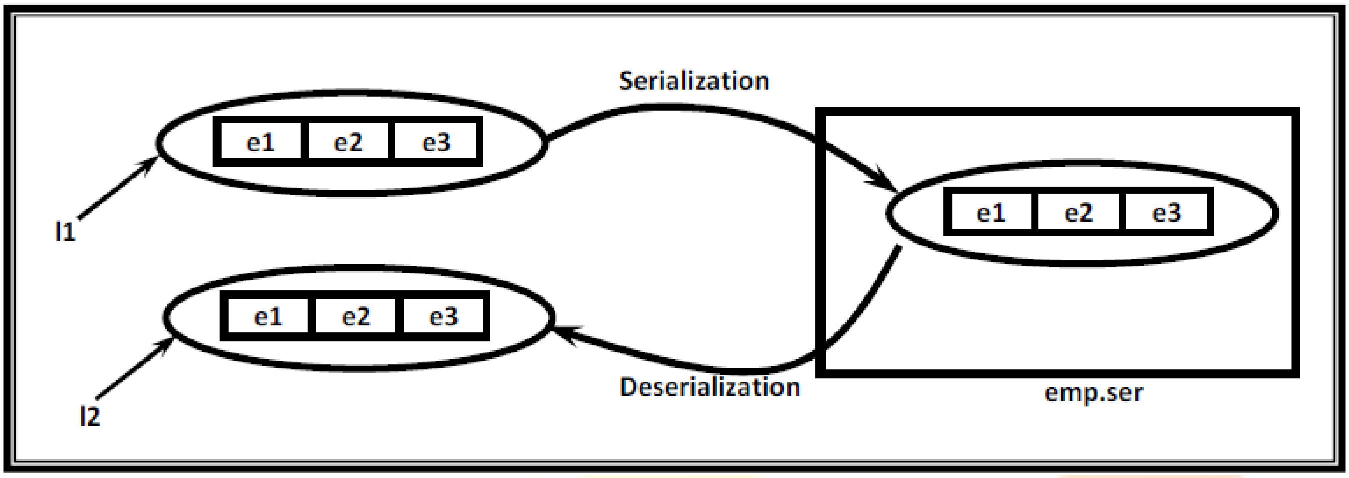
3. After creating the Map object, if we are trying to change the content(add|remove|replace elements)then we will get UnsupportedOperationException because Map is immutable(unmodifiable).

Eg:

```
Map<String,String> map=Map.ofEntries(entry("A","Apple"),entry("B","Banana"));
map.put("C","Cat"); → UnsupportedOperationException
map.remove("A"); → UnsupportedOperationException
```

Serialization for unmodifiable Collections:

The immutable collection objects are serializable iff all elements are serializable.
In Brief form, the process of writing state of an object to a file is called Serialization and the process of reading state of an object from the file is called Deserialization.



```
import java.util.*;
import java.io.*;
class Employee implements Serializable
{
   private int eno;
   private String ename;
   Employee(int eno,String ename)
   {
      this.eno=eno;
      this.ename=ename;
   }
   public String toString()
   {
      return String.format("%d=%s",eno,ename);
   }
}
```

```java
class Test
{
   public static void main(String[] args) throws Exception
   {
      Employee e1= new Employee(100,"Sunny");
      Employee e2= new Employee(200,"Bunny");
      Employee e3= new Employee(300,"Chinny");
      List<Employee> l1=List.of(e1,e2,e3);
      System.out.println(l1);

      System.out.println("Serialization of List Object...");
      FileOutputStream fos=new FileOutputStream("emp.ser");
      ObjectOutputStream oos=new ObjectOutputStream(fos);
      oos.writeObject(l1);

      System.out.println("Deserialization of List Object...");
      FileInputStream fis=new FileInputStream("emp.ser");
      ObjectInputStream ois=new ObjectInputStream(fis);
      List<Employee> l2=(List<Employee>)ois.readObject();
      System.out.println(l2);
      //l2.add(new Employee(400,"Vinnny"));//UnsupportedOperationException
   }
}
```

Output:

```
D:\durga_classes>java Test
[100=Sunny, 200=Bunny, 300=Chinny]
Serialization of List Object...
Deserialization of List Object...
[100=Sunny, 200=Bunny, 300=Chinny]
```

After deserialization also we cannot modify the content, otherwise we will get UnsupportedOperationException.

Note: The Factroy Methods introduced in Java 9 are not to create general collections and these are meant for creating immutable collections.

6.. Stream API Enhancements

In Java 9 as the part of Stream API, the following new methods introduced.

- takeWhile()
- dropWhile()
- Stream.iterate()
- Stream.ofNullable()

Note: takeWhile() and dropWhile() methods are default methods and iterate() and ofNullable() are static methods of Stream interface.

6.1 takeWhile()

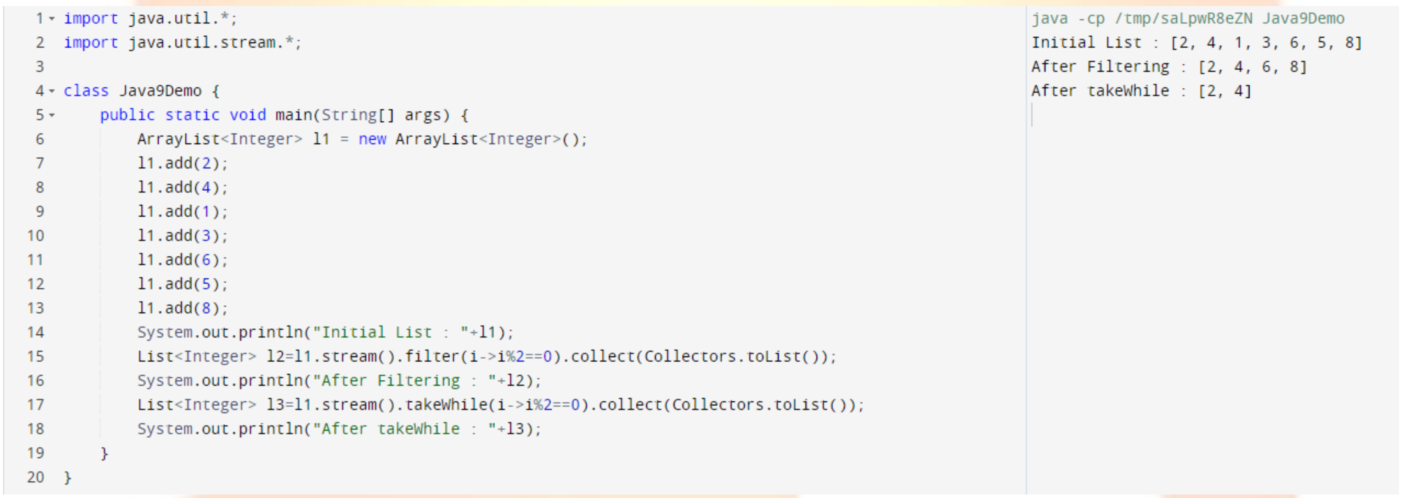
It is the default method present in Stream interface. It returns the stream of elements that matches the given predicate. It is similar to filter() method.

default Stream takeWhile(Predicate p)

Difference between takeWhile() and filter()

filter() method will process every element present in the stream and consider the element if predicate is true. But, in the case of takeWhile() method, there is no guarantee that it will process every element of the Stream. It will take elements from the 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.

Example: Take elements until we will get even numbers. Once we got odd number then stop and ignore rest of the stream.



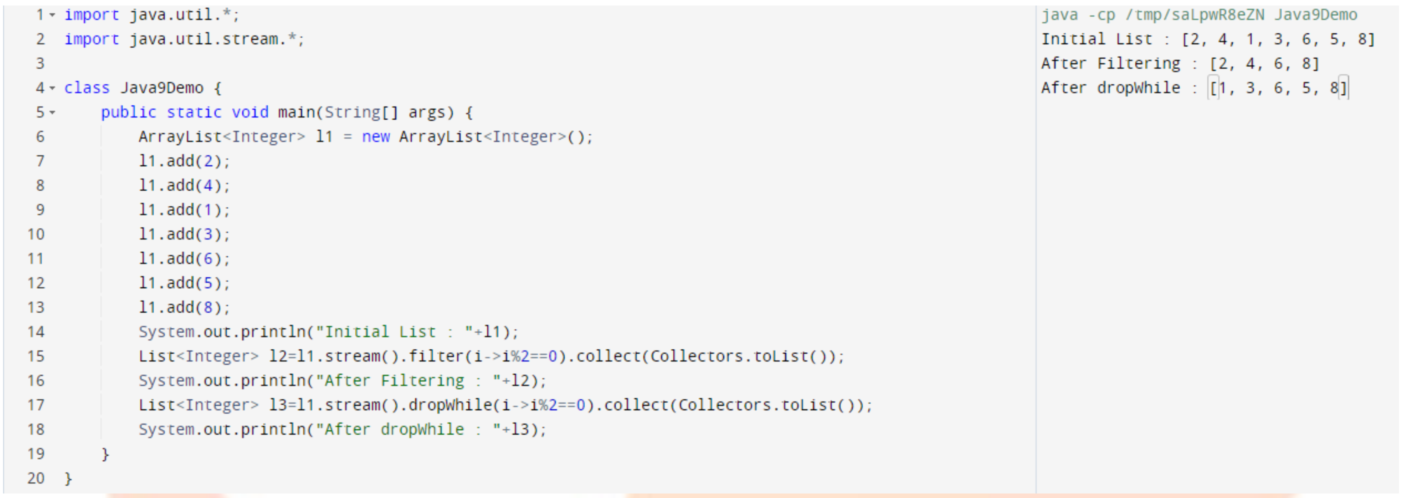
```java
import java.util.*;
import java.util.stream.*;

class Java9Demo {
    public static void main(String[] args) {
        ArrayList<Integer> l1 = new ArrayList<Integer>();
        l1.add(2);
        l1.add(4);
        l1.add(1);
        l1.add(3);
        l1.add(6);
        l1.add(5);
        l1.add(8);
        System.out.println("Initial List : "+l1);
        List<Integer> l2=l1.stream().filter(i->i%2==0).collect(Collectors.toList());
        System.out.println("After Filtering : "+l2);
        List<Integer> l3=l1.stream().takeWhile(i->i%2==0).collect(Collectors.toList());
        System.out.println("After takeWhile : "+l3);
    }
}
```

```
java -cp /tmp/saLpwR8eZN Java9Demo
Initial List : [2, 4, 1, 3, 6, 5, 8]
After Filtering : [2, 4, 6, 8]
After takeWhile : [2, 4]
```

6.2 dropWhile()

It is the default method present in Stream interface. It is the opposite of takeWhile() method. It drops elements instead of taking them as long as predicate returns true. Once predicate returns false then rest of the Stream will be returned.

default Stream dropWhile(Predicate p)



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

class Java9Demo {
    public static void main(String[] args) {
        ArrayList<Integer> l1 = new ArrayList<Integer>();
        l1.add(2);
        l1.add(4);
        l1.add(1);
        l1.add(3);
        l1.add(6);
        l1.add(5);
        l1.add(8);
        System.out.println("Initial List : "+l1);
        List<Integer> l2=l1.stream().filter(i->i%2==0).collect(Collectors.toList());
        System.out.println("After Filtering : "+l2);
        List<Integer> l3=l1.stream().dropWhile(i->i%2==0).collect(Collectors.toList());
        System.out.println("After dropWhile : "+l3);
    }
}
```

```
java -cp /tmp/saLpwR8eZN Java9Demo
Initial List : [2, 4, 1, 3, 6, 5, 8]
After Filtering : [2, 4, 6, 8]
After dropWhile : [1, 3, 6, 5, 8]
```

6.3 Stream.iterate()

It is the static method present in Stream interface.

Form-1: **iterate() method with 2 Arguments**.

This method introduced in Java 8.

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

It takes an initial value and a function that provides next value.

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

Output:

1
2
3
...infinite times

To limit the number of iterations we can use limit() method.

Eg: 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 argument iterate() method is there may be a chance of infinite loop. To avoid, we should use limit method. To prevent infinite loops, in Java 9, another version of iterate() method introduced, which is nothing but 3-arg iterate() method.

This method is something like for loop

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

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

This method takes an initial value, A terminate Predicate and A function that provides next value.

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

Output:

```
1
2
3
4
```

6.4 ofNullable()

This method will check whether the provided element is null or not. If it is not null, then this method returns the Stream of that element. If it is null then this method returns empty stream. This method is helpful to deal with null values in the stream. The main advantage of this method is to we can avoid NullPointerException and null checks everywhere. Usually we can use this method in flatMap() to handle null values.

public static Stream<T> ofNullable(T t)

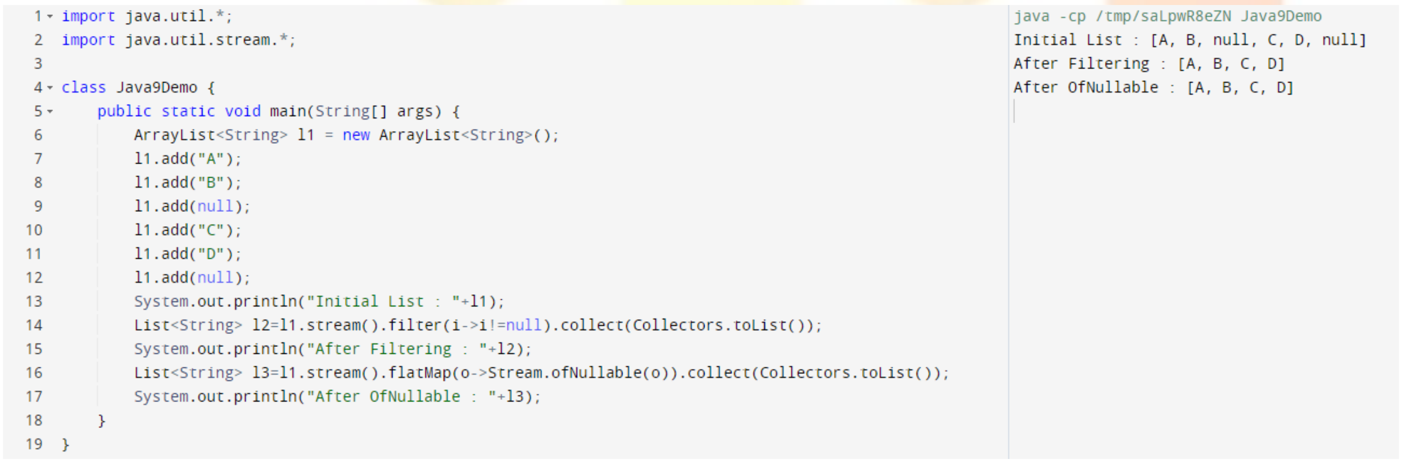
Eg 1:

```
List l=Stream.ofNullable(100).collect(Collectors.toList());
System.out.println(l);
```

Output:[100]

Eg 2:

```
List l=Stream.ofNullable(null).collect(Collectors.toList());
System.out.println(l);
```



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

class Java9Demo {
    public static void main(String[] args) {
        ArrayList<String> l1 = new ArrayList<String>();
        l1.add("A");
        l1.add("B");
        l1.add(null);
        l1.add("C");
        l1.add("D");
        l1.add(null);
        System.out.println("Initial List : "+l1);
        List<String> l2=l1.stream().filter(i->i!=null).collect(Collectors.toList());
        System.out.println("After Filtering : "+l2);
        List<String> l3=l1.stream().flatMap(o->Stream.ofNullable(o)).collect(Collectors.toList());
        System.out.println("After OfNullable : "+l3);
    }
}
```

```
java -cp /tmp/saLpwR8eZN Java9Demo
Initial List : [A, B, null, C, D, null]
After Filtering : [A, B, C, D]
After OfNullable : [A, B, C, D]
```

7.. JShell

JShell is also known as interactive console. JShell is Java's own **REPL** Tool. REPL means Read, Evaluate, Print and Loop. By using this tool, we can execute Java code snippets and we can get immediate results. For beginners it is very good to start programming in fun way. By using this JShell we can test and execute Java expressions, statements, methods, classes etc. It is useful for testing small code snippets very quickly, which can be plugged into our main coding based on our requirement.

Prior to Java 9 we cannot execute a single statement, expression, methods without full pledged classes. But in Java 9 with JShell we can execute any small piece of code without having complete class structure. It is new in Java but not a new concept. It is already there in other languages like Python, Swift, Lisp, Scala, Ruby etc. In Python it as available as **IDLE** while in Apple's Swift Programming Language it is available as **Playground**.

Limitations of JShell

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

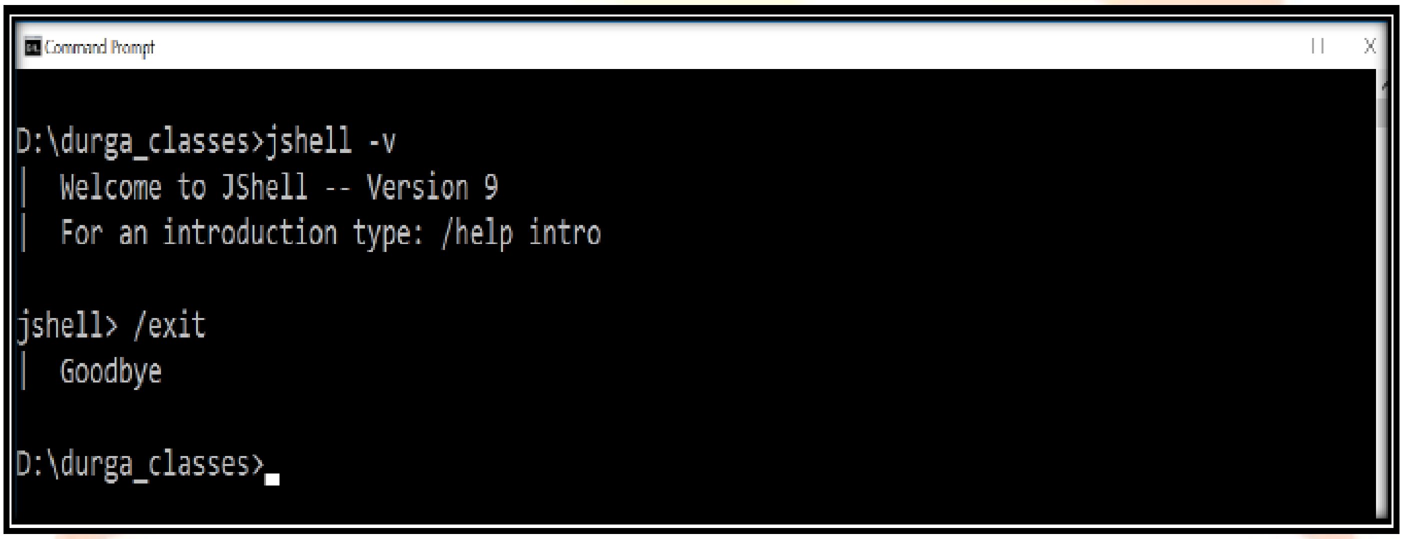
Starting and Stopping JShell

To Open the JShell from the command prompt in verbose mode we need below Command

jshell -v

To Exit the JShell:

jshell> /exit



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

jshell> /exit
|  Goodbye

D:\durga_classes>
```

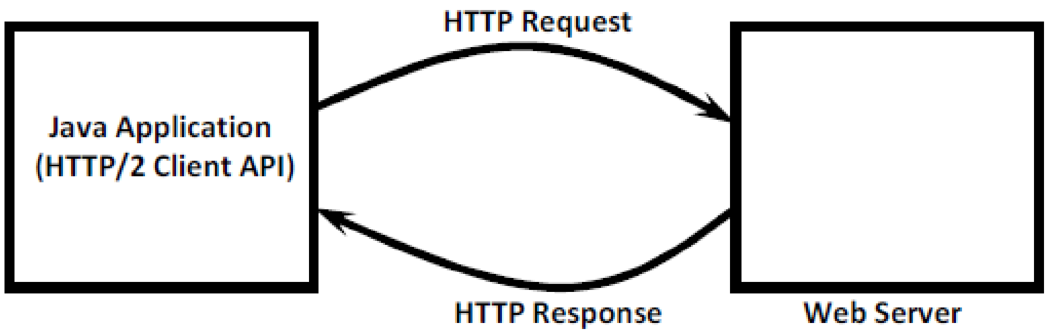
Everything what allowed in Java is a snippet. It can be Expression, Declaration, Statement, classes, interface, method, variable, import, etc. We can use all these as snippets from JShell. But package declarations are not allowed from the JShell.

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

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

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

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

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

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

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

jshell> m1()
hello
```

8.. HTTP/2 Client

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 Java 9, we are using **HttpURLConnection** class to send HTTP Request and to Process HTTP Response. It is the legacy class which was introduced as the part of JDK 1.1 (1997). There are several problems with this HttpURLConnection class.

- It is very difficult to use.
- It supports only HTTP/1.1 protocol but not HTTP/2(2015) where We can send only one request at a time per TCP Connection, which creates network traffic problems and performance problems and it supports only Text data but not binary data.
- It works only in Blocking Mode (Synchronous Mode), which creates performance problems.

Because of these problems, slowly developers started using third party Http Clients like Apache Http client and Google Http client etc. JDK 9 Engineers addresses these issues and introduced a brand-new HTTP/2 Client in Java 9.

Advantages of Java 9 HTTP/2 Client

- It is Lightweight and very easy to use.
- It supports both HTTP/1.1 and HTTP/2.
- It supports both Text data and Binary Data (Streams)
- It can work in both Blocking and Non-Blocking Modes (Synchronous Communication and Asynchronous Communication)
- It provides better performance and Scalability when compared with traditional HttpURLConnection.

Important Components of Java 9 HTTP/2 Client

In Java 9, HTTP/2 Client provided as incubator module.

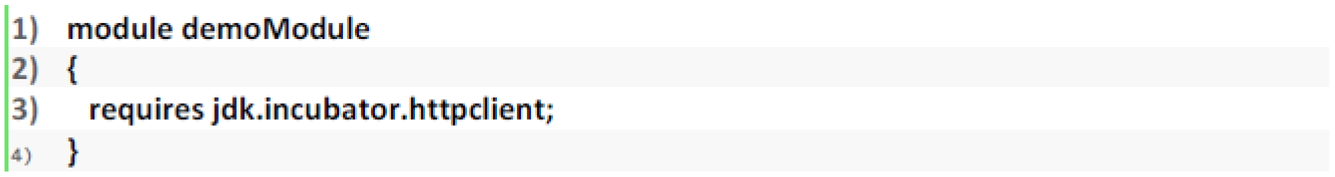
Module: jdk.incubator.httpclient

Package: jdk.incubator.http

Mainly 3 important classes are available:

- HttpClient
- HttpRequest
- 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 Http Request and process Http Response from Java Application

- Create HttpClient Object
- Create HttpRequest object
- Send HttpRequest by using HttpClient and Get the HttpResponse
- Process HttpResponse

1. Creation of 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();

2. Creation of HttpRequest object

We can create HttpRequest object as follows:

String url="http://www.durgasoft.com";

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

Note: newBuilder() method returns Builder object. GET() method sets the request method of this builder to GET. build() method builds and returns a HttpRequest.

public static HttpRequest.Builder newBuilder(URI uri)

public static HttpRequest.Builder GET()

public abstract HttpRequest build()

3.Send HttpRequest by using HttpClient and Get the HttpResponse

HttpClient contains the following methods:

- send() to send synchronous request(blocking mode)
- sendAsync() to send Asynchronous Request(Non Blocking Mode)

Example:

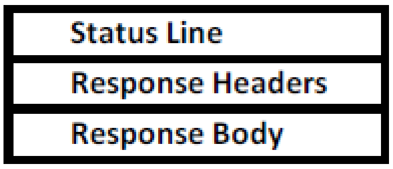
HttpResponse resp=client.send(req,HttpResponse.BodyHandler.asString());

HttpResponse resp=client.send(req,HttpResponse.BodyHandler.asFile(Paths.get("abc.txt")));

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

4. Process HttpResponse

HttpResponse contains the status code, response headers and body.



Structure of HTTP Response

HttpResponse class contains the following methods retrieve data from the response

- **statusCode()**
  Returns status code of the response. It may be (1XX,2XX,3XX,4XX,5XX)

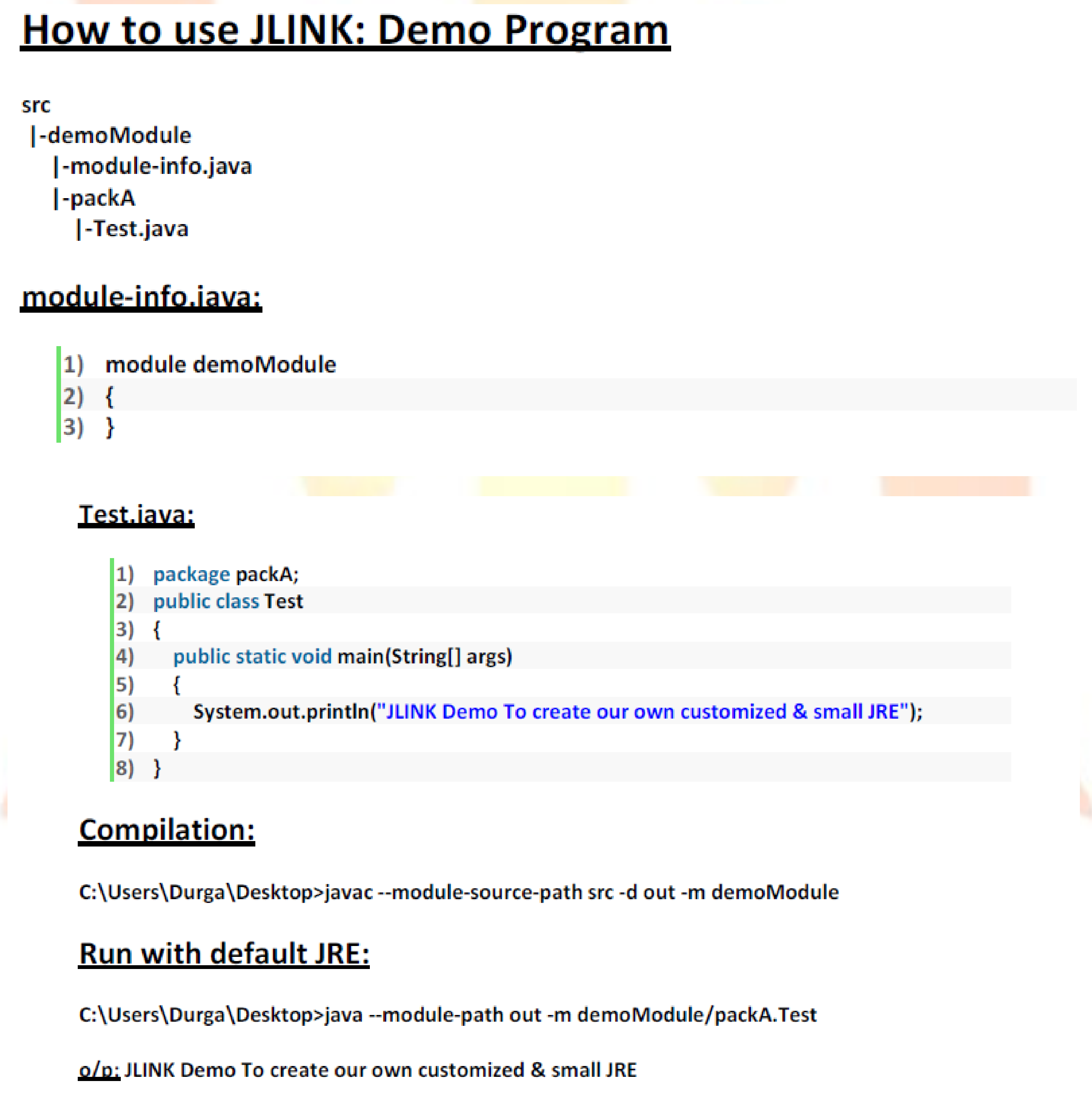
- **body()**
  Returns body of the response.

- **headers()**
  Returns header information of the response

9.. JLink (Java Linker)

Until 1.8 version to run a small Java program (like Hello World program) also, 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. (No one invite a bigger Elephant into their small house). 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 (which is 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). Now, our Custom JRE contains only required modules and classes instead of having all 4300+ classes. It reduces the size of Java Runtime Environment, which makes java best suitable for IOT and micro services. Hence, Jlink's main intention is to avoid shipping everything and, also, to run on very small devices with little memory. By using JLink, we can get our own very small JRE. JLink also has a list of plugins (like compress) that will help optimize our solutions.



How to use JLINK: Demo Program

```
src
 |-demoModule
    |-module-info.java
    |-packA
       |-Test.java
```

module-info.java:

```
module demoModule
{
}
```

Test.java:

```
package packA;
public class Test
{
   public static void main(String[] args)
   {
      System.out.println("JLINK Demo To create our own customized & small JRE");
   }
}
```

Compilation:

```
C:\Users\Durga\Desktop>javac --module-source-path src -d out -m demoModule
```

Run with default JRE:

```
C:\Users\Durga\Desktop>java --module-path out -m demoModule/packA.Test
```

o/p: JLINK Demo To create our own customized & small JRE

10.. Process API Updates

Until java 8, communicating 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 (i.e. os to os). For example, in windows one way, but in Mac other way. Being a programmer, we have to write code based on operating system, 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 Updates is the number 1 feature in Java 9.

With this Enhanced API, we can perform the following activities very easily.

- Get the Process ID (PID) of running process.
- Create a new process
- Destroy already running process
- Get the process handles for processes
- Get the parent and child processes of running process
- Get the process information like owner, children

What's New in Java 9 Process API

- Added several new methods (like pid(),info() etc) to **Process** class.
- Added several new methods (like startPipeline()) to **ProcessBuilder** class. We can use ProcessBuilder class to create operating system processes.
- Introduced a new powerful interface **ProcessHandle**. With this interface, we can access current running process, we can access parent and child processes of a particular process etc
- Introduced a new interface **ProcessHandle.Info**, by using this we can get complete information of a particular process.

Note: All these classes and interfaces are part of java.lang package and hence we are not required to use any import statement.

How to get ProcessHandle object

It is the most powerful and useful interface introduced in java 9.

We can get ProcessHandle object as follows

- **ProcessHandle handle=ProcessHandle.current();**
  Returns the ProcessHandle of current running Process

- **ProcessHandle handle=p.toHandle();**
  Returns the ProcessHandle of specified Process object.

- **Optional<ProcessHandle> handle=ProcessHandle.of(PID);**
  Returns the ProcessHandle of process with the specified pid. Here, the return type is Optional, because PID may exist or may not exist.

Use Case-1: To get the process ID (PID) of current process

```
public class Test
{
   public static void main(String[] args) throws Exception
   {
      ProcessHandle p=ProcessHandle.current();
      long pid=p.pid();
      System.out.println("The PID of current running JVM instance :"+pid);
      Thread.sleep(100000);
   }
}
```

We can see this process id in Task Manager (alt+ctrl+delete in windows).