



Java 8 & Java 9 New Features Study Material



Java 8 New Features

Table of Contents

1) Lambda Expressions	5
2) Functional Interfaces	7
3) Default methods	16
4) Predicates	21
5) Functions	24
6) Double colon operator (::)	26
7) Stream API	29
8) Date and Time API	35



Java 9 New Features

Table of Contents

1) Private Methods in Interfaces	40
2) Try With Resources Enhancements	47
3) Diamond Operator Enhancements	53
4) SafeVarargs Annotation Enhancements	60
5) Factory Methods for Creating unmodifiable Collections	64
6) Stream API Enhancements	71
7) The Java Shell (RPEL)	80
8) The Java Platform Module System (JPMS)	130
9) JLINK (JAVA LINKER)	176
10) Process API Updates	180
11) HTTP/2 Client	187



Java 8 New Features

Java 7 – July 28th 2011

2 Years 7 Months 18 Days

Java 8 - March 18th 2014

Java 9 - September 22nd 2016

Java 10 - 2018

After Java 1.5 version, Java 8 is the next major version.

Before Java 8, sun people gave importance only for objects but in 1.8 version oracle people gave the importance for functional aspects of programming to bring its benefits to Java. ie it doesn't mean Java is functional oriented programming language.

Java 8 New Features:

- 1) Lambda Expression
 - 2) Functional Interfaces
 - 3) Default methods
 - 4) Predicates
 - 5) Functions
 - 6) Double colon operator (::)
 - 7) Stream API
 - 8) Date and Time API
- Etc.....



Lambda (λ) Expressions

- ☀ Lambda calculus is a big change in mathematical world which has been introduced in 1930. Because of benefits of Lambda calculus slowly this concepts started using in programming world. "LISP" is the first programming which uses Lambda Expression.
- ☀ The other languages which uses lambda expressions are:
 - C#.Net
 - C Objective
 - C
 - C++
 - Python
 - Ruby etc.and finally in Java also.
- ☀ The Main Objective of Lambda Expression is to bring benefits of functional programming into Java.

What is Lambda Expression (λ):

- Lambda Expression is just an anonymous (nameless) function. That means the function which doesn't have the name, return type and access modifiers.
- Lambda Expression also known as anonymous functions or closures.

Ex: 1

```
public void m1() {  
    sop("hello");  
}  
    }  
    () → {  
        sop("hello");  
    }  
    () → { sop("hello"); }  
    () → sop("hello");
```

Ex:2

```
public void add(inta, int b) {  
    sop(a+b);  
}  
    }  
    (inta, int b) → sop(a+b);
```

- If the type of the parameter can be decided by compiler automatically based on the context then we can remove types also.
- The above Lambda expression we can rewrite as $(a,b) \rightarrow \text{sop}(a+b)$;



Ex: 3

```
public String str(String str) {  
    return str;  
}
```

(String str) → return str;
↓
(str) → str;

Conclusions:

- 1) A lambda expression can have zero or more number of parameters (arguments).

Ex:

```
() → sop("hello");  
(int a) → sop(a);  
(inta, int b) → return a+b;
```

- 2) Usually we can specify type of parameter. If the compiler expects the type based on the context then we can remove type. i.e., programmer is not required.

Ex:

```
(inta, int b) → sop(a+b);  
↓  
(a,b) → sop(a+b);
```

- 3) If multiple parameters present then these parameters should be separated with comma (,).
- 4) If zero number of parameters available then we have to use empty parameter [like ()].

Ex: () → sop("hello");

- 5) If only one parameter is available and if the compiler can expect the type then we can remove the type and parenthesis also.

Ex:

```
(int a) → sop(a);  
↓  
(a) → sop(a);  
↓  
A → sop(a);
```

- 6) Similar to method body lambda expression body also can contain multiple statements. If more than one statements present then we have to enclose inside within curly braces. If one statement present then curly braces are optional.
- 7) Once we write lambda expression we can call that expression just like a method, for this functional interfaces are required.



Functional Interfaces

If an interface contains only one abstract method, such type of interfaces are called functional interfaces and the method is called functional method or single abstract method (SAM).

Ex:

- | | | |
|-------------------|---|-------------------------------------|
| 1) Runnable | → | It contains only run() method |
| 2) Comparable | → | It contains only compareTo() method |
| 3) ActionListener | → | It contains only actionPerformed() |
| 4) Callable | → | It contains only call() method |

Inside functional interface in addition to single Abstract method (SAM) we write any number of default and static methods.

Ex:

```
1) interface Interf {  
2)     public abstract void m1();  
3)     default void m2() {  
4)         System.out.println ("hello");  
5)     }  
6) }
```

In Java 8, Sun Micro System introduced @Functional Interface annotation to specify that the interface is Functional Interface.

Ex:

```
@Functional Interface  
    Interface Interf {  
        public void m1();  
    }  
}
```

} This code compiles without any compilation errors.

Inside Functional Interface we can take only one abstract method, if we take more than one abstract method then compiler raises an error message that is called we will get compilation error.

Ex:

```
@Functional Interface {  
    public void m1();  
    public void m2();  
}
```

} This code gives compilation error.



Inside Functional Interface we have to take exactly only one abstract method. If we are not declaring that abstract method then compiler gives an error message.

Ex:

```
@Functional Interface {  
    interface Interface {  
    }  
}
```

 compilation error

Functional Interface with respect to Inheritance:

If an interface extends Functional Interface and child interface doesn't contain any abstract method then child interface is also Functional Interface

Ex:

```
1) @Functional Interface  
2) interface A {  
3)     public void methodOne();  
4) }  
5) @Functional Interface  
6) Interface B extends A {  
7) }
```

In the child interface we can define exactly same parent interface abstract method.

Ex:

```
1) @Functional Interface  
2) interface A {  
3)     public void methodOne();  
4) }  
5) @Functional Interface  
6) interface B extends A {  
7)     public void methodOne();  
8) }
```

 } No Compile Time Error

In the child interface we can't define any new abstract methods otherwise child interface won't be Functional Interface and if we are trying to use @Functional Interface annotation then compiler gives an error message.



```
1) @Functional Interface {  
2) interface A {  
3)     public void methodOne();  
4) }  
5) @Functional Interface  
6) interface B extends A {  
7)     public void methodTwo();  
8) }
```

} Compiletime Error

Ex:

```
@Functional Interface  
interface A {  
    public void methodOne();  
}  
interface B extends A {  
    public void methodTwo();  
}
```

} No compile time error

→ This's Normal interface so that code compiles without error

In the above example in both parent & child interface we can write any number of default methods and there are no restrictions. Restrictions are applicable only for abstract methods.

Functional Interface Vs Lambda Expressions:

Once we write Lambda expressions to invoke it's functionality, then Functional Interface is required. We can use Functional Interface reference to refer Lambda Expression.

Where ever Functional Interface concept is applicable there we can use Lambda Expressions

Ex:1 Without Lambda Expression

```
1) interface Interf {  
2)     public void methodOne() {}  
3)     public class Demo implements Interface {  
4)         public void methodOne() {  
5)             System.out.println("method one execution");  
6)         }  
7)     public class Test {  
8)         public static void main(String[] args) {  
9)             Interfi = new Demo();  
10)            i.methodOne();  
11)        }  
12) }
```



Above code With Lambda expression

```
1) interface Interf {
2)     public void methodOne() {}
3)     class Test {
4)         public static void main(String[] args) {
5)             Interfi = () → System.out.println("MethodOne Execution");
6)             i.methodOne();
7)         }
8)     }
```

Without Lambda Expression

```
1) interface Interf {
2)     public void sum(inta,int b);
3) }
4) class Demo implements Interf {
5)     public void sum(inta,int b) {
6)         System.out.println("The sum:"+(a+b));
7)     }
8) }
9) public class Test {
10)     public static void main(String[] args) {
11)         Interfi = new Demo();
12)         i.sum(20,5);
13)     }
14) }
```

Above code With Lambda Expression

```
1) interface Interf {
2)     public void sum(inta, int b);
3) }
4) class Test {
5)     public static void main(String[] args) {
6)         Interfi = (a,b) → System.out.println("The Sum:" +(a+b));
7)         i.sum(5,10);
8)     }
9) }
```



Without Lambda Expressions

```
1) interface Interf {  
2)     public int square(int x);  
3) }  
4) class Demo implements Interf {  
5)     public int square(int x) {  
6)         return x*x; OR (int x) → x*x  
7)     }  
8) }  
9) class Test {  
10)     public static void main(String[] args) {  
11)         Interfi = new Demo();  
12)         System.out.println("The Square of 7 is: " + i.square(7));  
13)     }  
14) }
```

Above code with Lambda Expression

```
1) interface Interf {  
2)     public int square(int x);  
3) }  
4) class Test {  
5)     public static void main(String[] args) {  
6)         Interfi = x → x*x;  
7)         System.out.println("The Square of 5 is:" + i.square(5));  
8)     }  
9) }
```

Without Lambda expression

```
1) class MyRunnable implements Runnable {  
2)     public void main() {  
3)         for(int i=0; i<10; i++) {  
4)             System.out.println("Child Thread");  
5)         }  
6)     }  
7) }  
8) class ThreadDemo {  
9)     public static void main(String[] args) {  
10)         Runnable r = new myRunnable();  
11)         Thread t = new Thread(r);  
12)         t.start();  
13)         for(int i=0; i<10; i++) {  
14)             System.out.println("Main Thread")  
15)         }  
16)     }
```



|17) }

With Lambda expression

```
1) class ThreadDemo {
2)     public static void main(String[] args) {
3)         Runnable r = () -> {
4)             for(int i=0; i<10; i++) {
5)                 System.out.println("Child Thread");
6)             }
7)         };
8)         Thread t = new Thread(r);
9)         t.start();
10)        for(i=0; i<10; i++) {
11)            System.out.println("Main Thread");
12)        }
13)    }
14) }
```

Anonymous inner classes vs Lambda Expressions

Wherever we are using anonymous inner classes there may be a chance of using Lambda expression to reduce length of the code and to resolve complexity.

Ex: With anonymous inner class

```
1) class Test {
2)     public static void main(String[] args) {
3)         Thread t = new Thread(new Runnable() {
4)             public void run() {
5)                 for(int i=0; i<10; i++) {
6)                     System.out.println("Child Thread");
7)                 }
8)             }
9)         });
10)        t.start();
11)        for(int i=0; i<10; i++)
12)            System.out.println("Main thread");
13)    }
14) }
```



With Lambda expression

```
1) class Test {  
2)     public static void main(String[] args) {  
3)         Thread t = new Thread(() → {  
4)                                     for(int i=0; i<10; i++) {  
5)                                         System.out.println("Child Thread");  
6)                                     }  
7)         });  
8)         t.start();  
9)         for(int i=0; i<10; i++) {  
10)             System.out.println("Main Thread");  
11)         }  
12)     }  
13) }
```

What are the advantages of Lambda expression?

- ☀ We can reduce length of the code so that readability of the code will be improved.
- ☀ We can resolve complexity of anonymous inner classes.
- ☀ We can provide Lambda expression in the place of object.
- ☀ We can pass lambda expression as argument to methods.

Note:

- ☀ Anonymous inner class can extend concrete class, can extend abstract class, can implement interface with any number of methods but
- ☀ Lambda expression can implement an interface with only single abstract method (Functional Interface).
- ☀ Hence if anonymous inner class implements Functional Interface in that particular case only we can replace with lambda expressions. Hence wherever anonymous inner class concept is there, it may not possible to replace with Lambda expressions.
- ☀ Anonymous inner class! = Lambda Expression
- ☀ Inside anonymous inner class we can declare instance variables.
- ☀ Inside anonymous inner class "this" always refers current inner class object (anonymous inner class) but not related outer class object

Ex:

- ☀ Inside lambda expression we can't declare instance variables.
- ☀ Whatever the variables declare inside lambda expression are simply acts as local variables
- ☀ Within lambda expression "this" keyword represents current outer class object reference (that is current enclosing class reference in which we declare lambda expression)



Ex:

```
1) interface Interf {
2)     public void m1();
3) }
4) class Test {
5)     int x = 777;
6)     public void m2() {
7)         Interfi = () -> {
8)             int x = 888;
9)             System.out.println(x); 888
10)            System.out.println(this.x); 777
11)        };
12)        i.m1();
13)    }
14)    public static void main(String[] args) {
15)        Test t = new Test();
16)        t.m2();
17)    }
18) }
```

- ☀ From lambda expression we can access enclosing class variables and enclosing method variables directly.
- ☀ The local variables referenced from lambda expression are implicitly final and hence we can't perform re-assignment for those local variables otherwise we get compile time error.

Ex:

```
1) interface Interf {
2)     public void m1();
3) }
4) class Test {
5)     int x = 10;
6)     public void m2() {
7)         int y = 20;
8)         Interfi = () -> {
9)             System.out.println(x); 10
10)            System.out.println(y); 20
11)            x = 888;
12)            y = 999; //CE
13)        };
14)        i.m1();
15)        y = 777;
16)    }
17)    public static void main(String[] args) {
18)        Test t = new Test();
19)        t.m2();
20)    }
21) }
```



Differences between anonymous inner classes and Lambda expression

Anonymous Inner class	Lambda Expression
It's a class without name	It's a method without name (anonymous function)
Anonymous inner class can extend Abstract and concrete classes	lambda expression can't extend Abstract and concrete classes
Anonymous inner class can implement An interface that contains any number of Abstract methods	lambda expression can implement an Interface which contains single abstract method (Functional Interface)
Inside anonymous inner class we can Declare instance variables.	Inside lambda expression we can't Declare instance variables, whatever the variables declared are simply acts as local variables.
Anonymous inner classes can be Instantiated	lambda expressions can't be instantiated
Inside anonymous inner class "this" Always refers current anonymous Inner class object but not outer class Object.	Inside lambda expression "this" Always refers current outer class object. That is enclosing class object.
Anonymous inner class is the best choice If we want to handle multiple methods.	Lambda expression is the best Choice if we want to handle interface With single abstract method (Functional Interface).
In the case of anonymous inner class At the time of compilation a separate Dot class file will be generated (outerclass\$1.class)	At the time of compilation no dot Class file will be generated for Lambda expression. It simply converts in to private method outer class.
Memory allocated on demand Whenever we are creating an object	Reside in permanent memory of JVM (Method Area).



Default Methods

- ☀ Until 1.7 version onwards inside interface we can take only public abstract methods and public static final variables (every method present inside interface is always public and abstract whether we are declaring or not).
- ☀ Every variable declared inside interface is always public static final whether we are declaring or not.
- ☀ But from 1.8 version onwards in addition to these, we can declare default concrete methods also inside interface, which are also known as defender methods.
- ☀ We can declare default method with the keyword "default" as follows

```
1) default void m1(){  
2) System.out.println ("Default Method");  
3) }
```

- ☀ Interface default methods are by-default available to all implementation classes. Based on requirement implementation class can use these default methods directly or can override.

Ex:

```
1) interface Interf {  
2)     default void m1() {  
3)         System.out.println("Default Method");  
4)     }  
5) }  
6) class Test implements Interf {  
7)     public static void main(String[] args) {  
8)         Test t = new Test();  
9)         t.m1();  
10)    }  
11) }
```

- ☀ Default methods also known as defender methods or virtual extension methods.
- ☀ The main advantage of default methods is without effecting implementation classes we can add new functionality to the interface (backward compatibility).

Note: We can't override object class methods as default methods inside interface otherwise we get compile time error.



Ex:

```
1) interface Interf {  
2)     default inthashCode() {  
3)         return 10;  
4)     }  
5) }
```

CompileTimeError

Reason: Object class methods are by-default available to every Java class hence it's not required to bring through default methods.

Default method vs multiple inheritance

Two interfaces can contain default method with same signature then there may be a chance of ambiguity problem (diamond problem) to the implementation class. To overcome this problem compulsory we should override default method in the implementation class otherwise we get compile time error.

```
1) Eg 1:  
2) interface Left {  
3)     default void m1() {  
4)         System.out.println("Left Default Method");  
5)     }  
6) }  
7)  
8) Eg 2:  
9) interface Right {  
10)     default void m1() {  
11)         System.out.println("Right Default Method");  
12)     }  
13) }  
14)  
15) Eg 3:  
16) class Test implements Left, Right {}
```



How to override default method in the implementation class?

In the implementation class we can provide complete new implementation or we can call any interface method as follows.

`interfacename.super.m1();`

Ex:

```
1) class Test implements Left, Right {  
2)     public void m1() {  
3)         System.out.println("Test Class Method"); OR Left.super.m1();  
4)     }  
5)     public static void main(String[] args) {  
6)         Test t = new Test();  
7)         t.m1();  
8)     }  
9) }
```

Differences between interface with default methods and abstract class

Even though we can add concrete methods in the form of default methods to the interface, it won't be equal to abstract class.

Interface with Default Methods	Abstract Class
Inside interface every variable is Always public static final and there is No chance of instance variables	Inside abstract class there may be a Chance of instance variables which Are required to the child class.
Interface never talks about state of Object.	Abstract class can talk about state of Object.
Inside interface we can't declare Constructors.	Inside abstract class we can declare Constructors.
Inside interface we can't declare Instance and static blocks.	Inside abstract class we can declare Instance and static blocks.
Functional interface with default Methods Can refer lambda expression.	Abstract class can't refer lambda Expressions.
Inside interface we can't override Object class methods.	Inside abstract class we can override Object class methods.

Interface with default method != abstract class



Static methods inside interface:

- ☀ From 1.8 version onwards in addition to default methods we can write static methods also inside interface to define utility functions.
- ☀ Interface static methods by-default not available to the implementation classes hence by using implementation class reference we can't call interface static
- ☀ methods. We should call interface static methods by using interface name.

Ex:

```
1) interface Interf {  
2)     public static void sum(int a, int b) {  
3)         System.out.println("The Sum:"+(a+b));  
4)     }  
5) }  
6) class Test implements Interf {  
7)     public static void main(String[] args) {  
8)         Test t = new Test();  
9)         t.sum(10, 20); //CE  
10)        Test.sum(10, 20); //CE  
11)        Interf.sum(10, 20);  
12)    }  
13) }
```

- ☀ As interface static methods by default not available to the implementation class, overriding concept is not applicable.
- ☀ Based on our requirement we can define exactly same method in the implementation class, it's valid but not overriding.

Ex:1

```
1) interface Interf {  
2)     public static void m1() {}  
3) }  
4) class Test implements Interf {  
5)     public static void m1() {}  
6) }
```

It's valid but not overriding



Ex:2

```
1) interface Interf {  
2)     public static void m1() {}  
3) }  
4) class Test implements Interf {  
5)     public void m1() {}  
6) }
```

This's valid but not overriding

Ex3:

```
1) class P {  
2)     private void m1() {}  
3) }  
4) class C extends P {  
5)     public void m1() {}  
6) }
```

This's valid but not overriding

From 1.8 version onwards we can write main() method inside interface and hence we can run interface directly from the command prompt.

Ex:

```
1) interface Interf {  
2)     public static void main(String[] args) {  
3)         System.out.println("Interface Main Method");  
4)     }  
5) }
```

At the command prompt:

Javac Interf.java

JavaInterf



Predicates

- A predicate is a function with a single argument and returns boolean value.
- To implement predicate functions in Java, Oracle people introduced Predicate interface in 1.8 version (i.e., Predicate<T>).
- Predicate interface present in *Java.util.function* package.
- It's a functional interface and it contains only one method i.e., test()

Ex:

```
interface Predicate<T>{  
    public boolean test(T t);  
}
```

As predicate is a functional interface and hence it can refer lambda expression

Ex:1 Write a predicate to check whether the given integer is greater than 10 or not.

Ex:

```
public boolean test(Integer l) {  
    if (l > 10) {  
        return true;  
    }  
    else {  
        return false;  
    }  
}
```



```
(Integer l) → {  
    if(l > 10)  
        return true;  
    else  
        return false;  
}
```



$l \rightarrow (l > 10);$



```
predicate<Integer> p = l → (l > 10);  
System.out.println (p.test(100)); true  
System.out.println (p.test(7)); false
```



Program:

```
1) import java.util.function;  
2) class Test {  
3)     public static void main(String[] args) {  
4)         predicate<Integer> p = i -> (i>10);  
5)         System.out.println(p.test(100));  
6)         System.out.println(p.test(7));  
7)         System.out.println(p.test(true)); //CE  
8)     }  
9) }
```

1 Write a predicate to check the length of given string is greater than 3 or not.

```
Predicate<String> p = s -> (s.length() > 3);  
System.out.println (p.test("rvkb")); true  
System.out.println (p.test("rk")); false
```

#-2 write a predicate to check whether the given collection is empty or not.

```
Predicate<collection> p = c -> c.isEmpty();
```

Predicate joining

It's possible to join predicates into a single predicate by using the following methods.

```
and()  
or()  
negate()
```

these are exactly same as logical AND ,OR complement operators

Ex:

```
1) import java.util.function.*;  
2) class test {  
3)     public static void main(string[] args) {  
4)         int[] x = {0, 5, 10, 15, 20, 25, 30};  
5)         predicate<integer> p1 = i->i>10;  
6)         predicate<integer> p2=i -> i%2==0;  
7)         System.out.println("The Numbers Greater Than 10:");  
8)         m1(p1, x);  
9)         System.out.println("The Even Numbers Are:");  
10)        m1(p2, x);  
11)        System.out.println("The Numbers Not Greater Than 10:");  
12)        m1(p1.negate(), x);  
13)        System.out.println("The Numbers Greater Than 10 And Even Are:â€œ");  
14)        m1(p1.and(p2), x);  
15)        System.out.println("The Numbers Greater Than 10 OR Even:â€œ");  
16)        m1(p1.or(p2), x);  
17)    }
```



```
18) public static void m1(predicate<integer>p, int[] x) {  
19)     for(int x1:x) {  
20)         if(p.test(x1))  
21)             System.out.println(x1);  
22)     }  
23) }  
24) }
```




Functions

- Functions are exactly same as predicates except that functions can return any type of result but function should (can) return only one value and that value can be any type as per our requirement.
- To implement functions oracle people introduced Function interface in 1.8 version.
- Function interface present in *Java.util.function* package.
- Functional interface contains only one method i.e., `apply()`

```
interface function(T,R) {  
    public R apply(T t);  
}
```

Assignment: Write a function to find length of given input string.

Ex:

```
1) import java.util.function.*;  
2) class Test {  
3)     public static void main(String[] args) {  
4)         Function<String, Integer> f = s -> s.length();  
5)         System.out.println(f.apply("Durga"));  
6)         System.out.println(f.apply("Soft"));  
7)     }  
8) }
```

Note: Function is a functional interface and hence it can refer lambda expression.



Differences between predicate and function

Predicate	Function
To implement conditional checks We should go for predicate	To perform certain operation And to return some result we Should go for function.
Predicate can take one type Parameter which represents Input argument type. Predicate<T>	Function can take 2 type Parameters. First one represent Input argument type and Second one represent return Type. Function<T,R>
Predicate interface defines only one method called test()	Function interface defines only one Method called apply().
public boolean test(T t)	public R apply(T t)
Predicate can return only boolean value.	Function can return any type of value

Note: Predicate is a boolean valued function and(), or(), negate() are default methods present inside Predicate interface.



Method And Constructor References by using :: (Double Colon) Operator

- Functional Interface method can be mapped to our specified method by using :: (double colon) operator. This is called method reference.
- Our specified method can be either static method or instance method.
- Functional Interface method and our specified method should have same argument types, except this the remaining things like return type, methodname, modifiers etc are not required to match.

Syntax

If our specified method is static method
`Classname::methodName`

If the method is instance method
`Objref::methodName`

- Functional Interface can refer lambda expression and Functional Interface can also refer method reference. Hence lambda expression can be replaced with method reference. Hence method reference is alternative syntax to lambda expression.

Ex: With Lambda Expression

```
1) class Test {  
2)     public static void main(String[] args) {  
3)         Runnable r = () -> {  
4)             for(int i=0; i<=10; i++) {  
5)                 System.out.println("Child Thread");  
6)             }  
7)         };  
8)         Thread t = new Thread(r);  
9)         t.start();  
10)        for(int i=0; i<=10; i++) {  
11)            System.out.println("Main Thread");  
12)        }  
13)    }  
14) }
```

With Method Reference

```
1) class Test {  
2)     public static void m1() {  
3)         for(int i=0; i<=10; i++) {  
4)             System.out.println("Child Thread");  
5)         }  
6)     }
```



```
7) public static void main(String[] args) {
8)     Runnable r = Test:: m1;
9)     Thread t = new Thread(r);
10)    t.start();
11)    for(int i=0; i<=10; i++) {
12)        System.out.println("Main Thread");
13)    }
14) }
```

In the above example Runnable interface run() method referring to Test class static method m1().
Method reference to Instance method:

Ex:

```
1) interface Interf {
2)     public void m1(int i);
3) }
4) class Test {
5)     public void m2(int i) {
6)         System.out.println("From Method Reference:"+i);
7)     }
8)     public static void main(String[] args) {
9)         Interf f = l -> sop("From Lambda Expression:"+i);
10)        f.m1(10);
11)        Test t = new Test();
12)        Interf i1 = t::m2;
13)        i1.m1(20);
14)    }
15) }
```

In the above example functional interface method m1() referring to Test class instance method m2().

The main advantage of method reference is we can use already existing code to implement functional interfaces (code reusability).

Constructor References

We can use :: (double colon) operator to refer constructors also

Syntax: classname :: new

Ex:

```
Interf f = sample :: new;
functional interface f referring sample class constructor
```



Ex:

```
1) class Sample {  
2)     private String s;  
3)     Sample(String s) {  
4)         this.s = s;  
5)         System.out.println("Constructor Executed:"+s);  
6)     }  
7) }  
8) interface Interf {  
9)     public Sample get(String s);  
10) }  
11) class Test {  
12)     public static void main(String[] args) {  
13)         Interf f = s -> new Sample(s);  
14)         f.get("From Lambda Expression");  
15)         Interf f1 = Sample :: new;  
16)         f1.get("From Constructor Reference");  
17)     }  
18) }
```

Note: In method and constructor references compulsory the argument types must be matched.



Streams

To process objects of the collection, in 1.8 version Streams concept introduced.

What is the differences between Java.util.streams and Java.io streams?

java.util streams meant for processing objects from the collection. I.e, it represents a stream of objects from the collection but Java.io streams meant for processing binary and character data with respect to file. I.e it represents stream of binary data or character data from the file .hence Java.io streams and Java.util streams both are different.

What is the difference between collection and stream?

- If we want to represent a group of individual objects as a single entity then We should go for collection.
- If we want to process a group of objects from the collection then we should go for streams.
- We can create a stream object to the collection by using stream() method of Collection interface. stream() method is a default method added to the Collection in 1.8 version.

default Stream stream()

Ex: Stream s = c.stream();

- Stream is an interface present in java.util.stream. Once we got the stream, by using that we can process objects of that collection.
- We can process the objects in the following 2 phases

1.Configuration

2.Processing

1) Configuration:

We can configure either by using filter mechanism or by using map mechanism.

Filtering:

We can configure a filter to filter elements from the collection based on some boolean condition by using `filter()` method of Stream interface.

```
public Stream filter(Predicate<T> t)
```

here (Predicate<T> t) can be a boolean valued function/lambda expression

Ex:

```
Stream s = c.stream();  
Stream s1 = s.filter(i -> i%2==0);
```

Hence to filter elements of collection based on some Boolean condition we should go for filter() method.

Mapping:

If we want to create a separate new object, for every object present in the collection based on our requirement then we should go for `map()` method of `Stream` interface.

```
public Stream map (Function f);
```

It can be lambda expression also

Ex:

```
Stream s = c.stream();
Stream s1 = s.map(i-> i+10);
```

Once we performed configuration we can process objects by using several methods.

2) Processing

- processing by collect() method
- Processing by count()method
- Processing by sorted()method
- Processing by min() and max() methods
- forEach() method
- toArray() method
- Stream.of()method



Processing by collect() method

This method collects the elements from the stream and adding to the specified to the collection indicated (specified) by argument.

Ex 1: To collect only even numbers from the array list

Approach-1: Without Streams

```
1) import Java.util.*;
2) class Test {
3)     public static void main(String[] args) {
4)         ArrayList<Integer> l1 = new ArrayList<Integer>();
5)         for(int i=0; i<=10; i++) {
6)             l1.add(i);
7)         }
8)         System.out.println(l1);
9)         ArrayList<Integer> l2 = new ArrayList<Integer>();
10)        for(Integer i:l1) {
11)            if(i%2 == 0)
12)                l2.add(i);
13)        }
14)        System.out.println(l2);
15)    }
16) }
```

Approach-2: With Streams

```
1) import Java.util.*;
2) import Java.util.stream.*;
3) class Test {
4)     public static void main(String[] args) {
5)         ArrayList<Integer> l1 = new ArrayList<Integer>();
6)         for(int i=0; i<=10; i++) {
7)             l1.add(i);
8)         }
9)         System.out.println(l1);
10)        List<Integer> l2 = l1.stream().filter(i -> i%2==0).collect(Collectors.toList());
11)        System.out.println(l2);
12)    }
13) }
```




Ex: Program for map() and collect() Method

```
1) import Java.util.*;
2) import Java.util.stream.*;
3) class Test {
4)     public static void main(String[] args) {
5)         ArrayList<String> l = new ArrayList<String>();
6)         l.add("rvk"); l.add("rk"); l.add("rkv"); l.add("rvki"); l.add("rvkir");
7)         System.out.println(l);
8)         List<String> l2 = l.stream().map(s -
9)             >s.toUpperCase()).collect(Collectors.toList());
10)         System.out.println(l2);
11)     }
```

II.Processing by count()method

This method returns number of elements present in the stream.

```
public long count()
```

Ex:

```
long count = l.stream().filter(s ->s.length()==5).count();
sop("The number of 5 length strings is:"+count);
```

III.Processing by sorted()method

If we sort the elements present inside stream then we should go for sorted() method. the sorting can either default natural sorting order or customized sorting order specified by comparator.

sorted()- default natural sorting order

sorted(Comparator c)-customized sorting order.

Ex:

```
List<String> l3=l.stream().sorted().collect(Collectors.toList());
sop("according to default natural sorting order:"+l3);
```

```
List<String> l4=l.stream().sorted((s1,s2) -> -s1.compareTo(s2)).collect(Collectors.toList());
sop("according to customized sorting order:"+l4);
```



IV.Processing by min() and max() methods

`min(Comparator c)`

returns minimum value according to specified comparator.

`max(Comparator c)`

returns maximum value according to specified comparator

Ex:

```
String min=l.stream().min((s1,s2) -> s1.compareTo(s2)).get();  
sop("minimum value is:"+min);
```

```
String max=l.stream().max((s1,s2) -> s1.compareTo(s2)).get();  
sop("maximum value is:"+max);
```

V.forEach() method

This method will not return anything.

This method will take lambda expression as argument and apply that lambda expression for each element present in the stream.

Ex:

```
l.stream().forEach(s->sop(s));  
l3.stream().forEach(System.out::println);
```

Ex:

```
1) import Java.util.*;  
2) import Java.util.stream.*;  
3) class Test1 {  
4)     public static void main(String[] args) {  
5)         ArrayList<Integer> l1 = new ArrayList<Integer>();  
6)         l1.add(0); l1.add(15); l1.add(10); l1.add(5); l1.add(30); l1.add(25); l1.add(20);  
7)         System.out.println(l1);  
8)         ArrayList<Integer> l2=l1.stream().map(i-> i+10).collect(Collectors.toList());  
9)         System.out.println(l2);  
10)        long count = l1.stream().filter(i->i%2==0).count();  
11)        System.out.println(count);  
12)        List<Integer> l3=l1.stream().sorted().collect(Collectors.toList());  
13)        System.out.println(l3);  
14)        Comparator<Integer> comp=(i1,i2)->i1.compareTo(i2);  
15)        List<Integer> l4=l1.stream().sorted(comp).collect(Collectors.toList());  
16)        System.out.println(l4);  
17)        Integer min=l1.stream().min(comp).get();  
18)        System.out.println(min);  
19)        Integer max=l1.stream().max(comp).get();  
20)        System.out.println(max);
```



```
21)         l3.stream().forEach(i->sop(i));  
22)         l3.stream().forEach(System.out:: println);  
23)  
24)     }  
25) }
```

VI.toArray() method

We can use toArray() method to copy elements present in the stream into specified array

```
Integer[] ir = l1.stream().toArray(Integer[] :: new);  
for(Integer i: ir) {  
    sop(i);  
}
```

VII.Stream.of()method

We can also apply a stream for group of values and for arrays.

Ex:

```
Stream s=Stream.of(99,999,9999,99999);  
s.forEach(System.out:: println);
```

```
Double[] d={10.0,10.1,10.2,10.3};  
Stream s1=Stream.of(d);  
s1.forEach(System.out :: println);
```



Date and Time API: (Joda-Time API)

Until Java 1.7 version the classes present in Java.util package to handle Date and Time (like Date, Calendar, TimeZone etc) are not up to the mark with respect to convenience and performance.

To overcome this problem in the 1.8 version oracle people introduced Joda-Time API. This API developed by joda.org and available in Java in the form of Java.time package.

program for to display System Date and time.

```
1) import java.time.*;
2) public class DateTime {
3)     public static void main(String[] args) {
4)         LocalDate date = LocalDate.now();
5)         System.out.println(date);
6)         LocalTime time = LocalTime.now();
7)         System.out.println(time);
8)     }
9) }
```

O/p:

2015-11-23

12:39:26:587

Once we get LocalDate object we can call the following methods on that object to retrieve Day, month and year values separately.

Ex:

```
1) import java.time.*;
2) class Test {
3)     public static void main(String[] args) {
4)         LocalDate date = LocalDate.now();
5)         System.out.println(date);
6)         int dd = date.getDayOfMonth();
7)         int mm = date.getMonthValue();
8)         int yy = date.getYear();
9)         System.out.println(dd+"..." +mm+"..." +yy);
10)        System.out.printf("\n%d-%d-%d", dd, mm, yy);
11)    }
12) }
```

Once we get LocalTime object we can call the following methods on that object.



Ex:

```
1) import java.time.*;
2) class Test {
3)     public static void main(String[] args) {
4)         LocalDateTime time = LocalDateTime.now();
5)         int h = time.getHour();
6)         int m = time.getMinute();
7)         int s = time.getSecond();
8)         int n = time.getNano();
9)         System.out.printf("\n%d:%d:%d:%d", h, m, s, n);
10)    }
11) }
```

If we want to represent both Date and Time then we should go for LocalDateTime object.

```
LocalDateTime dt = LocalDateTime.now();
System.out.println(dt);
```

O/p: 2015-11-23T12:57:24.531

We can represent a particular Date and Time by using LocalDateTime object as follows.

Ex:

```
LocalDateTime dt1 = LocalDateTime.of(1995, Month.APRIL, 28, 12, 45);
sop(dt1);
```

Ex:

```
LocalDateTime dt1 = LocalDateTime.of(1995, 04, 28, 12, 45);
Sop(dt1);
Sop("After six months:" + dt.plusMonths(6));
Sop("Before six months:" + dt.minusMonths(6));
```

To Represent Zone:

ZoneId object can be used to represent Zone.

Ex:

```
1) import java.time.*;
2) class ProgramOne {
3)     public static void main(String[] args) {
4)         ZoneId zone = ZoneId.systemDefault();
5)         System.out.println(zone);
6)     }
7) }
```



We can create ZoneId for a particular zone as follows

Ex:

```
ZoneId la = ZoneId.of("America/Los_Angeles");
ZonedDateTime zt = ZonedDateTime.now(la);
System.out.println(zt);
```

Period Object:

Period object can be used to represent quantity of time

Ex:

```
LocalDate today = LocalDate.now();
LocalDate birthday = LocalDate.of(1989,06,15);
Period p = Period.between(birthday,today);
System.out.printf("age is %d year %d months %d
days",p.getYears(),p.getMonths(),p.getDays());
```

write a program to check the given year is leap year or not

```
1) import java.time.*;
2) public class Leapyear {
3)     int n = Integer.parseInt(args[0]);
4)     Year y = Year.of(n);
5)     if(y.isLeap())
6)         System.out.printf("%d is Leap year",n);
7)     else
8)         System.out.printf("%d is not Leap year",n);
9) }
```



Java 9

New Features



Table of Contents

12) Private Methods in Interfaces	40
13) Try With Resources Enhancements	47
14) Diamond Operator Enhancements	53
15) SafeVarargs Annotation Enhancements	60
16) Factory Methods for Creating unmodifiable Collections	64
17) Stream API Enhancements	71
18) The Java Shell (RPEL)	80
19) The Java Platform Module System (JPMS)	130
20) JLINK (JAVA LINKER)	176
21) Process API Updates	180
22) HTTP/2 Client	187



Private Methods in Interfaces

Need Of Default Methods inside interfaces:

Prior to Java 8, Every method present inside interface is always public and abstract whether we are declaring or not.

```
1) interface Prior2Java8Interf
2) {
3)     public void m1();
4)     public void m2();
5) }
```

Assume this interface is implemented by 1000s of classes and each class provided implementation for both methods.

```
1) interface Prior2Java8Interf
2) {
3)     public void m1();
4)     public void m2();
5) }
6) class Test1 implements Prior2Java8Interf
7) {
8)     public void m1(){}
9)     public void m2(){}
10) }
11) class Test2 implements Prior2Java8Interf
12) {
13)     public void m1(){}
14)     public void m2(){}
15) }
16) class Test3 implements Prior2Java8Interf
17) {
18)     public void m1(){}
19)     public void m2(){}
20) }
21) class Test1000 implements Prior2Java8Interf
22) {
23)     public void m1(){}
24)     public void m2(){}
25) }
```

It is valid because all implementation classes provided implementation for both m1() and m2().



Assume our programming requirement is we have to extend the functionality of this interface by adding a new method m3().

```
1) interface Prior2Java8Interf
2) {
3)     public void m1();
4)     public void m2();
5)     public void m3();
6) }
```

If we add new method m3() to the interface then all the implementation classes will be effected and won't be compiled, because every implementation class should implement all methods of interface.

```
1) class Test1 implements Prior2Java8Interf
2) {
3)     public void m1(){ }
4)     public void m2(){ }
5) }
```

CE: Test1 is not abstract and does not override abstract method m3() in PriorJava8Interf

Hence prior to java 8, it is impossible to extend the functionality of an existing interface without effecting implementation classes. JDK 8 Engineers addresses this issue and provides solution in the form of Default methods, which are also known as Defender methods or Virtual Extension Methods.

How to Declare Default Methods inside interfaces:

In Java 8, inside interface we can define default methods with implementation as follows.

```
1) interface Java8Interf
2) {
3)     public void m1();
4)     public void m2();
5)     default void m3()
6)     {
7)         // "Default Implementation
8)     }
9) }
```

Interface default methods are by-default available to all implementation classes. Based on requirement implementation class can ignore these methods or use these default methods directly or can override.



Hence the main advantage of Default Methods inside interfaces is, without effecting implementation classes we can extend functionality of interface by adding new methods (Backward compatibility).

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).

Eg:

```
1) public interface Java8DBLogging
2) {
3)     //Abstract Methods List
4)     default void logInfo(String message)
5)     {
6)         Step1: Connect to DataBase
7)         Setp2: Log Info Message
8)         Setp3: Close the DataBase connection
9)     }
10)    default void logWarn(String message)
11)    {
12)        Step1: Connect to DataBase
13)        Setp2: Log Warn Message
14)        Setp3: Close the DataBase connection
15)    }
16)    default void logError(String message)
17)    {
18)        Step1: Connect to DataBase
19)        Setp2: Log Error Message
20)        Setp3: Close the DataBase connection
21)    }
22)    default void logFatal(String message)
23)    {
24)        Step1: Connect to DataBase
25)        Setp2: Log Fatal Message
26)        Setp3: Close the DataBase connection
27)    }
28) }
```

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 separate that common code into a private method and we can call that private method from every default method which required that functionality.

```
1) public interface Java9DBLogging
2) {
3)     //Abstract Methods List
4)     default void logInfo(String message)
5)     {
6)         log(message,"INFO");
7)     }
8)     default void logWarn(String message)
9)     {
10)        log(message,"WARN");
11)    }
12)    default void logError(String message)
13)    {
14)        log(message,"ERROR");
15)    }
16)    default void logFatal(String message)
17)    {
18)        log(message,"FATAL");
19)    }
20)    private void log(String msg,String logLevel)
21)    {
22)        Step1: Connect to DataBase
23)        Step2: Log Message with the Provided logLevel
24)        Step3: Close the DataBase Connection
25)    }
26) }
```

Demo Program for private instance methods inside interface:

private instance methods will provide code reusability for default methods.

```
1) interface Java9Interf
2) {
3)     default void m1()
4)     {
5)         m3();
6)     }
7)     default void m2()
8)     {
9)         m3();
10)    }
```



```
11) private void m3()
12) {
13)     System.out.println("common functionality of methods m1 & m2");
14) }
15) }
16) class Test implements Java9Interf
17) {
18)     public static void main(String[] args)
19)     {
20)         Test t = new Test();
21)         t.m1();
22)         t.m2();
23)         //t.m3(); ==>CE
24)     }
25) }
```

Output:

```
D:\java9durga>java Test
common functionality of methods m1 & m2
common functionality of methods m1 & m2
```

Inside Java 8 interfaces, we can take public static methods also. If several static methods having some common functionality, we can separate 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.

Demo Program for private static methods:

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

```
1) interface Java9Interf
2) {
3)     public static void m1()
4)     {
5)         m3();
6)     }
7)     public static void m2()
8)     {
9)         m3();
10)    }
11)    private static void m3()
12)    {
13)        System.out.println("common functionality of methods m1 & m2");
14)    }
15) }
16) class Test implements Java9Interf
17) {
```



```
18) public static void main(String[] args)
19) {
20)     Java9Interf.m1();
21)     Java9Interf.m2();
22) }
23) }
```

Output:

D:\durga_classes>java Test

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:

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.

Note:

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

JDK 7 vs JDK 8 vs JDK9:

1. Prior to java 8, we can declare only public-abstract methods and public-static-final variables inside interfaces.

```
1) interface Prior2Java8Interface
2) {
3)     public-static-final variables
4)     public-abstract methods
5) }
```

2. In Java 8, we can declare default and public-static methods also inside interface.

```
1) interface Java8Interface
2) {
3)     public-static-final variables
4)     public-abstract methods
```



- 5) default methods with implementation
- 6) public static methods with implementation
- 7) }

3. In Java 9, We can declare private instance and private static methods also inside interface.

- 1) interface Java9Interface
- 2) {
- 3) public-static-final variables
- 4) public-abstract methods
- 5) default methods with implementation
- 6) public static methods with implementation
- 7) private instance methods with implementation
- 8) private static methods with implementation
- 9) }

Note: The main advantage of private methods inside interface is Code Reusability without effecting implementation classes.



Try with Resources Enhancements

Need of Try with Resources:

Until 1.6 version it is highly recommended to write finally block to close all resources which are open as part of try block.

```
1) BufferedReader br=null;
2) try
3) {
4)   br=new BufferedReader(new FileReader("abc.txt"));
5)   //use br based on our requirements
6) }
7) catch(IOException e)
8) {
9)   // handling code
10) }
11) finally
12) {
13)   if(br != null)
14)     br.close();
15) }
```

Problems in this Approach:

1. Compulsory programmer is required to close all opened resources which increases the complexity of the programming
2. Compulsory we should write finally block explicitly which increases length of the code and reduces readability.

To overcome these problems Sun People introduced "try with resources" in 1.7 version.

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 the complexity of programming will be reduced. It is not required to write finally block explicitly and hence length of the code will be reduced and readability will be improved.

```
1) try(BufferedReader br=new BufferedReader(new FileReader("abc.txt")))
2) {
3)   use be based on our requirement, br will be closed automatically , Once control reaches
   end of try either normally or abnormally and we are not required to close explicitly
```




```
4) }  
5) catch(IOException e)  
6) {  
7)    // handling code  
8) }
```

Conclusions:

1. We can declare any number of resources but all these resources should be separated with ; (semicolon)

```
1) try(R1 ; R2 ; R3)  
2) {  
3)    -----  
4) }
```

2. All resources should be AutoCloseable resources. A resource is said to be auto closable if and only if the corresponding class implements the *java.lang.AutoCloseable* interface either directly or indirectly.

All database 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.

3. AutoCloseable interface introduced in Java 1.7 Version and it contains only one method: close()

4. All resource reference variables should be final or effectively final and hence we can't perform reassignment within the try block.

```
1) try(BufferedReader br=new BufferedReader(new FileReader("abc.txt")))  
2) {  
3)    br=new BufferedReader(new FileReader("abc.txt"));  
4) }
```

CE : Can't reassign a value to final variable br

5. Untill 1.6 version try should be followed by either catch or finally but in 1.7 version we can take only try with resource without catch or finally

```
1) try(R)  
2) {  
3)    //valid  
4) }
```

6. The main advantage of "try with resources" is finally block will become dummy because we are not required to close resources of explicitly.



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.

```
1) BufferedReader br=new BufferedReader(new FileReader("abc.txt"))
2) try(br)
3) {
4)    // Risky code
5) }
```

This syntax is invalid in until java 1.8V.

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.

Solution-1: Creation of Resource in try block primary list

```
1) try(BufferedReader br=new BufferedReader(new FileReader("abc.txt")))
2) {
3)    // It is valid syntax in java 1.8V
4) }
```

Solution-2: Assign resource with new reference variable

```
1) BufferedReader br=new BufferedReader(new FileReader("abc.txt"))
2) try(BufferedReader br1=br)
3) {
4)    // It is valid syntax in java 1.8V
5) }
```

But from JDK 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.

```
1) BufferedReader br=new BufferedReader(new FileReader("abc.txt"))
2) try(br)
3) {
4)    // It is valid in JDK 9 but in valid until JDK 1.8V
5) }
```

we make sure resource(br) 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.



```
1) MyResource r1 = new MyResource();
2) MyResource r2 = new MyResource();
3) MyResource r3 = new MyResource();
4) try(r1,r2,r3)
5) {
6) }
```

Demo Program:

```
1) class MyResource implements AutoCloseable
2) {
3)     MyResource()
4)     {
5)         System.out.println("Resource Creation...");
6)     }
7)     public void doProcess()
8)     {
9)         System.out.println("Resource Processing...");
10)    }
11) }
12) public void close()
13) {
14)     System.out.println("Resource Closing...");
15) }
16) }
17) class Test
18) {
19)     public static void preJDK7()
20)     {
21)         MyResource r=null;
22)         try
23)         {
24)             r=new MyResource();
25)             r.doProcess();
26)         }
27)         catch (Exception e)
28)         {
29)             System.out.println("Handling:"+e);
30)         }
31)         finally
32)         {
33)             try
34)             {
35)                 if (r!=null)
36)                 {
37)                     r.close();
38)                 }
39)             }
```



```
40)     catch (Exception e)
41)     {
42)         System.out.println("Handling:"+e);
43)     }
44) }
45) }
46) public static void JDK7()
47) {
48)     try(MyResource r=new MyResource())
49)     {
50)         r.doProcess();
51)     }
52)     catch(Exception e)
53)     {
54)         System.out.println("Handling:"+e);
55)     }
56) }
57) public static void JDK9()
58) {
59)     MyResource r= new MyResource();
60)     try(r)
61)     {
62)         r.doProcess();
63)     }
64)     catch(Exception e)
65)     {
66)         System.out.println("Handling:"+e);
67)     }
68) }
69) public static void multipleJDK9()
70) {
71)     MyResource r1= new MyResource();
72)     MyResource r2= new MyResource();
73)     MyResource r3= new MyResource();
74)     MyResource r4= new MyResource();
75)     try(r1;r2;r3;r4)
76)     {
77)         r1.doProcess();
78)         r2.doProcess();
79)         r3.doProcess();
80)         r4.doProcess();
81)     }
82)     catch(Exception e)
83)     {
84)         System.out.println("Handling:"+e);
85)     }
86) }
87) public static void main(String[] args)
```



```
88) {  
89)     System.out.println("Program Execution With PreJDK7");  
90)     preJDK7();  
91)  
92)     System.out.println("Program Execution With JDK7");  
93)     JDK7();  
94)  
95)     System.out.println("Program Execution With JDK9");  
96)     JDK9();  
97)     System.out.println("Program Execution Multiple Resources With JDK9");  
98)     multipleJDK9();  
99) }  
100) }
```

Output:

Program Execution With PreJDK7

Resource Creation...

Resource Processing...

Resource Closing...

Program Execution With JDK7

Resource Creation...

Resource Processing...

Resource Closing...

Program Execution With JDK9

Resource Creation...

Resource Processing...

Resource Closing...

Program Execution Multiple Resources With JDK9

Resource Creation...

Resource Creation...

Resource Creation...

Resource Creation...

Resource Processing...

Resource Processing...

Resource Processing...

Resource Processing...

Resource Closing...

Resource Closing...

Resource Closing...

Resource Closing...



Diamond Operator Enhancements

This enhancement is as the part of Milling Project Coin (JEP 213).

Before understanding this enhancement, we should aware Generics concept, which has been introduced in java 1.5 version.

The main objectives of Generics are:

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

Case 1: Type-Safety

Arrays are always type safe. i.e we can give the guarantee for the type of elements present inside array. For example if our programming requirement is to hold String type of objects, it is recommended to use String array. For the string array we can add only string type of objects. By mistake if we are trying to add any other type we will get compile time error.

Eg:

- 1) `String[] s = new String[100];`
- 2) `s[0] = "Durga";`
- 3) `s[1] = "Pavan";`
- 4) `s[2] = new Integer(10);` //error: incompatible types: Integer cannot be converted to String

`String[]` can contains only String type of elements. Hence, we can always give guarantee for the type of elements present inside array. Due to this arrays are safe to use with respect to type. Hence arrays are type safe.

But collections are not type safe that is we can't give any guarantee for the type of elements present inside collection. For example if our programming requirement is to hold only string type of objects, and if we choose `ArrayList`, by mistake if we are trying to add any other type we won't get any compile time error but the program may fail at runtime.

- 1) `ArrayList l = new ArrayList();`
- 2) `l.add("Durga");`
- 3) `l.add("Pavan");`
- 4) `l.add(new Integer(10));`
- 5)
- 6) `String name1=(String)l.get(0);`
- 7) `String name2=(String)l.get(1);`
- 8) `String name3=(String)l.get(2);` //RE: java.lang.ClassCastException

Hence we can't give any guarantee for the type of elements present inside collections. Due to this collections are not safe to use with respect to type, i.e collections are not Type-Safe.



Case 2: Type-Casting

In the case of array at the time of retrieval it is not required to perform any type casting.

- 1) `String[] s = new String[100];`
- 2) `s[0] = "Durga";`
- 3) `s[1] = "Pavan";`
- 4) ...
- 5) `String name1=s[0];`//Type casting is not required.

But in the case of collection at the time of retrieval compulsory we should perform type casting otherwise we will get compile time error.

Eg:

- 1) `ArrayList l = new ArrayList();`
- 2) `l.add("Durga");`
- 3) `l.add("Pavan");`
- 4)
- 5) `String name1=l.get(0);`//error: incompatible types: Object cannot be converted to String
- 6) `String name1=(String)l.get(0);`//valid

Hence in Collections Type Casting is bigger headache.

To overcome the above problems of collections(type-safety, type casting),Java people introduced Generics concept in 1.5version.Hence the main objectives of Generics are:

1. To provide Type Safety to the collections.
2. To resolve Type casting problems.

Example for Generic Collection:

To hold only string type of objects, we can create a generic version of ArrayList as follows.

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

Here String is called Parameter Type.

For this ArrayList we can add only string type of objects. By mistake if we are trying to add any other type then we will get compile time error.

- 1) `l.add("Durga");`//valid
- 2) `l.add("Kavi");`//valid
- 3) `l.add(new Integer(10));`//error: no suitable method found for add(Integer)

Hence, through generics we are getting type safety.

At the time of retrieval it is not required to perform any type casting we can assign elements directly to string type variables.



`String name1 = l.get(0);` //valid and Type Casting is not required.

Hence, through generic syntax we can resolve type casting problems.

Difference between Generic and Non-Generic Collections

<code>ArrayList l = new ArrayList();</code>	<code>ArrayList<String> l = new ArrayList<String>();</code>
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 is not Type-Safe.	For this ArrayList we can add only String type of Objects. By mistake if we are trying to add any other type, we will get compile time error.
At the time of retrieval compulsory we should perform Type casting.	At the time of retrieval, we are not required to perform Type casting.

Java 7 Diamond Operator (<>):

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.

Eg 2:

```
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, we can use.



Usage of Diamond Operator for Anonymous Classes:

In JDK 9, Usage of Diamond Operator extended to Anonymous classes also.

Anonymous class:

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

Eg 1:

```
1) Thread t = new Thread()  
2) {  
3) };
```

We are creating a child class that extends Thread class without name(Anonymous class) and we are creating object for that child class.

Eg 2:

```
1) Runnable r = new Runnable()  
2) {  
3) };
```

We are creating an implementation class for Runnable interface without name(Anonymous class) and we are creating object for that implementation class.

Eg 3:

```
1) ArrayList<String> l = new ArrayList<String>()  
2) {  
3) };
```

We are creating a child class that extends ArrayList class without name(Anonymous class) and we are creating object for that child class.

From JDK 9 onwards we can use Diamond Operator for Anonymous Classes also.

```
1) ArrayList<String> l = new ArrayList<>()  
2) {  
3) };
```

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



Demo Program - 1: To demonstrate usage of diamond operator for Anonymous Class

```
1) class MyGenClass<T>
2) {
3)     T obj;
4)     public MyGenClass(T obj)
5)     {
6)         this.obj = obj;
7)     }
8)
9)     public T getObj()
10)    {
11)        return obj;
12)    }
13)    public void process()
14)    {
15)        System.out.println("Processing obj...");
16)    }
17) }
18) public class Test
19) {
20)     public static void main(String[] args)
21)     {
22)         MyGenClass<String> c1 = new MyGenClass<String>("Durga")
23)         {
24)             public void process()
25)             {
26)                 System.out.println("Processing... " + getObj());
27)             }
28)         };
29)         c1.process();
30)
31)         MyGenClass<String> c2 = new MyGenClass<>("Pavan")
32)         {
33)             public void process()
34)             {
35)                 System.out.println("Processing... " + getObj());
36)             }
37)         };
38)         c2.process();
39)     }
40) }
```

Output:

Processing... Durga
Processing... Pavan



If we compile the above program according to Java 1.8 version, then we will get compile time error

D:\durga_classes>javac -source 1.8 Test.java

error: cannot infer type arguments for MyGenClass<T>

```
MyGenClass<String> c2 = new MyGenClass<>("Pavan")
```

^

reason: cannot use '<>' with anonymous inner classes in -source 1.8

(use -source 9 or higher to enable '<>' with anonymous inner classes)

Demo Program - 2: To demonstrate usage of diamond operator for Anonymous Class

```
1) import java.util.Iterator;
2) import java.util.NoSuchElementException;
3) public class DiamondOperatorDemo
4) {
5)     public static void main(String[] args)
6)     {
7)         String[] animals = { "Dog", "Cat", "Rat", "Tiger", "Elephant" };
8)         Iterator<String> iter = new Iterator<>()
9)         {
10)             int i = 0;
11)             public boolean hasNext()
12)             {
13)                 return i < animals.length;
14)             }
15)             public String next()
16)             {
17)                 if (!hasNext())
18)                     throw new NoSuchElementException();
19)                 return animals[i++];
20)             }
21)         };
22)         while (iter.hasNext())
23)         {
24)             System.out.println(iter.next());
25)         }
26)     }
27) }
```

Output:

Dog
Cat
Rat
Tiger
Elephant



Note - 1:

```
1) ArrayList<String> preJava7 = new ArrayList<String>();  
2) ArrayList<String> java7 = new ArrayList<>();  
3) ArrayList<String> java9 = new ArrayList<>()  
4) {  
5) };
```

Note - 2:

Be Ready for Partial Diamond Operator in the next versions of Java, as the part of Project "Amber". Open JDK people already working on this.

Eg: new Test<String, _>();



SafeVarargs Annotation Enhancements

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.

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

What is var-arg method?

Until 1.4 version, we can't declared 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 are called var-arg methods.

```
1) public class Test
2) {
3)     public static void m1(int... x)
4)     {
5)         System.out.println("var-arg method");
6)     }
7)     public static void main(String[] args)
8)     {
9)         m1();
10)        m1(10);
11)        m1(10,20,30);
12)    }
13) }
```

Output

var-arg method
var-arg method
var-arg method

Internally var-arg parameter will be converted into array.

```
1) public class Test
2) {
3)     public static void sum(int... x)
4)     {
5)         int total=0;
6)         for(int x1 : x)
7)         {
```



```
8)      total=total+x1;
9)      }
10)     System.out.println("The Sum:" + total);
11)     }
12)     public static void main(String[] args)
13)     {
14)         sum();
15)         sum(10);
16)         sum(10,20,30);
17)     }
18) }
```

Output

The Sum:0

The Sum:10

The Sum:60

Var-arg method with Generic Type:

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 ClassCastException. This problem is called Heap Pollution.

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

```
1) import java.util.*;
2) public class Test
3) {
4)     public static void main(String[] args)
5)     {
6)         List<String> l1= Arrays.asList("A","B");
7)         List<String> l2= Arrays.asList("C","D");
8)         m1(l1,l2);
9)     }
10)    public static void m1(List<String>... l)//argument will become List<String>[]
11)    {
12)        Object[] a = l;// we can assign List[] to Object[]
13)        a[0]=Arrays.asList(10,20);
14)        String name=(String)l[0].get(0);//String type pointing to Integer type
15)        System.out.println(name);
16)    }
17) }
```

Compilation:

javac Test.java

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

Execution:

```
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 name is trying to point to Integer type, which causes Heap Pollution and results ClassCastException.

```
String name = (String)l[0].get(0);
```

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.

```
1) import java.util.*;  
2) public class Test  
3) {  
4)     public static void main(String[] args)  
5)     {  
6)         List<String> l1= Arrays.asList("A","B");  
7)         List<String> l2= Arrays.asList("C","D");  
8)         m1(l1,l2);  
9)     }  
10) @SafeVarargs  
11) public static void m1(List<String>... l)  
12) {  
13)     for(List<String> l1: l)  
14)     {  
15)         System.out.println(l1);  
16)     }  
17) }  
18) }
```



Output:

[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.

Java 9 Enhancements to @SafeVarargs Annotation:

@SafeVarargs Annotation introduced in Java 7.

Until Java 8, this annotation is applicable only for static methods, final methods and constructors. But from Java 9 onwards, we can also use for private instance methods also.

```
1) import java.util.*;
2) public class Test
3) {
4)     @SafeVarargs //valid
5)     public Test(List<String>... l)
6)     {
7)     }
8)     @SafeVarargs //valid
9)     public static void m1(List<String>... l)
10)    {
11)    }
12)    @SafeVarargs //valid
13)    public final void m2(List<String>... l)
14)    {
15)    }
16)    @SafeVarargs //valid in Java 9 but not in Java 8
17)    private void m3(List<String>... l) {
18)    }
19) }
```

javac -source 1.8 Test.java

error: Invalid SafeVarargs annotation. Instance method m3(List<String>...) is not final.

```
    private void m3(List<String>... l)
           ^
```

javac -source 1.9 Test.java

We won't get any compile time error.

FAQs:

Q1. For which purpose we can use @SafeVarargs annotation?

Q2. What is Heap Pollution?



Factory Methods for creating unmodifiable Collections

List: An indexed Collection of elements where duplicates are allowed and insertion order is preserved.

Set: An unordered Collection of elements where duplicates are not allowed and insertion order is not preserved.

Map: A Map is a collection of key-value pairs and each key-value pair is called Entry. Entry is an inner interface present inside Map interface. Duplicate keys are not allowed, but values can be duplicated.

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

```
1) List<String> beers=new ArrayList<String>();  
2) beers.add("KF");  
3) beers.add("FO");  
4) beers.add("RC");  
5) beers.add("FO");  
6) beers =Collections.unmodifiableList(beers);
```

Eg 2: Creation of unmodifiable Set Object

```
1) Set<String> beers=new HashSet<String>();  
2) beers.add("KF");  
3) beers.add("KO");  
4) beers.add("RC");  
5) beers.add("FO");  
6) beers =Collections.unmodifiableSet(beers);
```

Eg 3: Creation of unmodifiable Map object

```
1) Map<String,String> map=new HashMap<String,String>();  
2) map.put("A","Apple");  
3) map.put("B","Banana");  
4) map.put("C","Cat");  
5) map.put("D","Dog");  
6) 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 address this problem and introduced several factory methods for creating unmodifiable collections.

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

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)

Upto 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 upto 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.

Eg:

```
1) import java.util.*;
2) class Test
3) {
4)     public static void main(String[] args)
```



```
5) {  
6)     Map.Entry<String,String> e1=Map.entry("A","Apple");  
7)     Map.Entry<String,String> e2=Map.entry("B","Banana");  
8)     Map.Entry<String,String> e3=Map.entry("C","Cat");  
9)     Map<String,String> m=Map.ofEntries(e1,e2,e3);  
10)    System.out.println(m);  
11) }  
12) }
```

In Short way we can also create as follows.

```
1) import static java.util.Map.entry;  
2) Map<String,String> map=Map.ofEntries(entry("A","Apple"),entry("B","Banana"),entry("C",  
    ,"Cat"),entry("D","Dog"));
```

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:

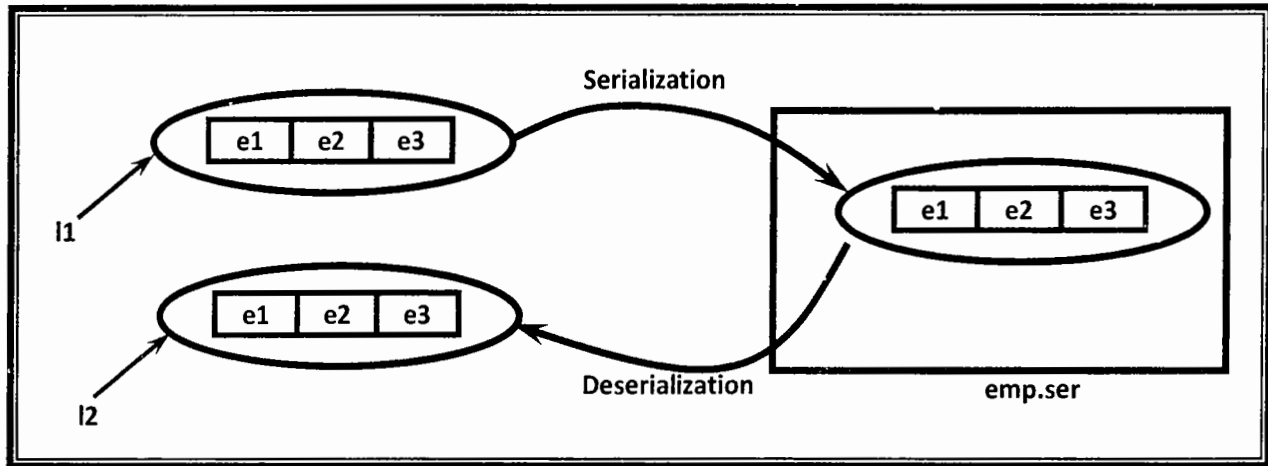
```
Map<String,String> map=Map.ofEntries(entry("A","Apple"),entry("B","Banana"));  
map.put("C","Cat"); → UnsupportedOperationException  
map.remove("A"); → UnsupportedOperationException
```



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**.



```
1) import java.util.*;
2) import java.io.*;
3) class Employee implements Serializable
4) {
5)     private int eno;
6)     private String ename;
7)     Employee(int eno,String ename)
8)     {
9)         this.eno=eno;
10)        this.ename=ename;
11)    }
12)    public String toString()
13)    {
14)        return String.format("%d=%s",eno,ename);
15)    }
16) }
17) class Test
18) {
19)     public static void main(String[] args) throws Exception
20)     {
21)         Employee e1= new Employee(100,"Sunny");
22)         Employee e2= new Employee(200,"Bunny");
23)         Employee e3= new Employee(300,"Chinny");
24)         List<Employee> l1=List.of(e1,e2,e3);
25)         System.out.println(l1);
26)
27)         System.out.println("Serialization of List Object...");
28)         FileOutputStream fos=new FileOutputStream("emp.ser");
```



```
29) ObjectOutputStream oos=new ObjectOutputStream(fos);
30) oos.writeObject(l1);
31)
32) System.out.println("Deserialization of List Object...");
33) FileInputStream fis=new FileInputStream("emp.ser");
34) ObjectInputStream ois=new ObjectInputStream(fis);
35) List<Employee> l2=(List<Employee>)ois.readObject();
36) System.out.println(l2);
37) //l2.add(new Employee(400,"Vinnny")); //UnsupportedOperationException
38) }
39) }
```

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 Factory Methods introduced in Java 9 are not to create general collections and these are meant for creating immutable collections.



Stream API Enhancements

Streams concept has been introduced in Java 1.8 version.

The main objective of Streams concept is to process elements of Collection with Functional Programming (Lambda Expressions).

What is the difference between java.util streams and java.io streams?

java.util streams meant for processing objects from the collection. ie. it represents a stream of objects from the collection but java.io streams meant for processing binary and character data with respect to file. i.e it represents stream of binary data or character data from the file. Hence java.io streams and java.util streams are different concepts.

What is the difference between Collections and Streams?

If we want to represent a group of individual objects as a single entity, then we should go for collection. If we want to process a group of objects from the collection then we should go for Streams.

How to Create Stream Object?

We can create stream object to the collection by using stream() method of Collection interface. stream() method is a default method added to the Collection in 1.8 version.

default Stream stream()

Ex: Stream s = c.stream(); // c is any Collection object

Note: Stream is an interface present in java.util.stream package.

How to process Objects of Collection By using Stream:

Once we got the stream, by using that we can process objects of that collection. For this we can use either filter() method or map() method.

Processing Objects by using filter() method:

We can use filter() method to filter elements from the collection based on some boolean condition.

public Stream filter(Predicate<T> t)

Here (Predicate<T> t) can be a boolean valued function/lambda expression



Demo Program:

```
1) import java.util.*;
2) import java.util.stream.*;
3) public class Test
4) {
5)     public static void main(String[] args)
6)     {
7)         ArrayList<Integer> l1 = new ArrayList<Integer>();
8)         for(int i =0; i <=10; i++)
9)         {
10)             l1.add(i);
11)         }
12)         System.out.println("Before Filtering:"+l1);
13)         List<Integer> l2=l1.stream().filter(i->i%2==0).collect(Collectors.toList());
14)         System.out.println("After Filtering:"+l2);
15)     }
16) }
```

Output:

Before Filtering:[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

After Filtering:[0, 2, 4, 6, 8, 10]

Processing Objects by using map() method:

If we want to create a separate new object, for every object present in the collection based on our requirement then we should go for map() method of Stream interface.

`public Stream map (Function f);`

The argument can be lambda expression also

Demo Program:

```
1) import java.util.*;
2) import java.util.stream.*;
3) public class Test
4) {
5)     public static void main(String[] args) {
6)         ArrayList<Integer> l1 = new ArrayList<Integer>();
7)         for(int i =0; i <=10; i++)
8)         {
9)             l1.add(i);
10)        }
11)        System.out.println("Before using map() method:"+l1);
12)        List<Integer> l2=l1.stream().map(i->i*i).collect(Collectors.toList());
13)        System.out.println("After using map() method:"+l2);
14)    }
15) }
```



Output:

Before using map() method:[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

After using map() method:[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

Processing Objects by using flatMap() method:

Both map and flatMap can be applied to a Stream<T> and they both return a Stream<R>. The difference is that the map operation produces one output value for each input value, whereas the flatMap operation produces an arbitrary number (zero or more) values for each input value.

Typical use is for the mapper function of flatMap to return Stream.empty() if it wants to send zero values, or something like Stream.of(x, y, z) if it wants to return several values.

Demo Program:

```
1) import java.util.*;
2) import java.util.stream.*;
3) public class Test
4) {
5)     public static void main(String[] args)
6)     {
7)         ArrayList<Integer> l1 = new ArrayList<Integer>();
8)         for(int i =0; i <=10; i++)
9)         {
10)            l1.add(i);
11)        }
12)        System.out.println("Before using map() method:"+l1);
13)        List<Integer> l2=l1.stream().flatMap(
14)            i->{ if (i%2 !=0) return Stream.empty();
15)                else return Stream.of(i);
16)            }).collect(Collectors.toList());
17)        System.out.println("After using flatMap() method:"+l2);
18)
19)        List<Integer> l3=l1.stream().flatMap(
20)            i->{ if (i%2 !=0) return Stream.empty();
21)                else return Stream.of(i,i*i);
22)            }).collect(Collectors.toList());
23)        System.out.println("After using flatMap() method:"+l3);
24)    }
25) }
```

Output:

D:\durga_classes>java Test

Before using map() method:[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

After using flatMap() method:[0, 2, 4, 6, 8, 10]

After using flatMap() method:[0, 0, 2, 4, 4, 16, 6, 36, 8, 64, 10, 100]

Q. What is the difference between map() and flatMap() methods?



Java 9 Enhancements for Stream API:

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

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

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

1. takeWhile():

It is the default method present in Stream interface.

default Stream takeWhile(Predicate p)

It returns the stream of elements that matches the given predicate.
It is similar to filter() method.

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.

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

Demo Program:

```
1) import java.util.*;
2) import java.util.stream.*;
3) public class Test
4) {
5)     public static void main(String[] args)
6)     {
7)         ArrayList<Integer> l1 = new ArrayList<Integer>();
8)         l1.add(2);
```



```
9)    l1.add(4);
10)   l1.add(1);
11)   l1.add(3);
12)   l1.add(6);
13)   l1.add(5);
14)   l1.add(8);
15)   System.out.println("Initial List:"+l1);
16)   List<Integer> l2=l1.stream().filter(i->i%2==0).collect(Collectors.toList());
17)   System.out.println("After Filtering:"+l2);
18)   List<Integer> l3=l1.stream().takeWhile(i->i%2==0).collect(Collectors.toList());
19)   System.out.println("After takeWhile:"+l3);
20)   }
21) }
```

Output:

Initial List:[2, 4, 1, 3, 6, 5, 8]

After Filtering:[2, 4, 6, 8]

After takeWhile:[2, 4]

Eg 2:

```
Stream.of("A", "AA", "BBB", "CCC", "CC", "C").takeWhile(s-
>s.length()<=2).forEach(System.out::println);
```

Output:

A

AA

2. dropWhile()

It is the default method present in Stream interface.

~~default Stream~~ dropWhile(Predicate p)

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.

Demo Program:

```
1) import java.util.*;
2) import java.util.stream.*;
3) public class Test
4) {
5)     public static void main(String[] args)
6)     {
7)         ArrayList<Integer> l1 = new ArrayList<Integer>();
```



```
8)    l1.add(2);
9)    l1.add(4);
10)   l1.add(1);
11)   l1.add(3);
12)   l1.add(6);
13)   l1.add(5);
14)   l1.add(8);
15)   System.out.println("Initial List:"+l1);
16)   List<Integer> l2=l1.stream().dropWhile(i->i%2==0).collect(Collectors.toList());
17)   System.out.println("After dropWhile:"+l2);
18)   }
19)   }
```

Output:

Initial List:[2, 4, 1, 3, 6, 5, 8]

After dropWhile:[1, 3, 6, 5, 8]

Eg 2:

```
Stream.of("A","AA","BBB","CCC","CC","C").dropWhile(s-
>s.length()<=2).forEach(System.out::println);
```

Output:

BBB
CCC
CC
C

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



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

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
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



4. ofNullable():

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

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.

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);
```

Output: []

Demo Program:

```
1) import java.util.*;  
2) import java.util.stream.*;  
3) import java.util.*;  
4) import java.util.stream.*;  
5) public class Test  
6) {  
7)     public static void main(String[] args)  
8)     {  
9)         List<String> l=new ArrayList<String>();  
10)        l.add("A");  
11)        l.add("B");  
12)        l.add(null);  
13)        l.add("C");  
14)        l.add("D");  
15)        l.add(null);  
16)        System.out.println(l);  
17)    }  
18)    List<String> l2= l.stream().filter(o->o!=null).collect(Collectors.toList());  
19)    System.out.println(l2);  
20)
```



```
21) List<String> l3= l.stream()  
22)           .flatMap(o->Stream.ofNullable(o)).collect(Collectors.toList());  
23) System.out.println(l3);  
24) }  
25) }
```

Output:

[A, B, null, C, D, null]
[A, B, C, D]
[A, B, C, D]

Demo Program:

```
1) import java.util.*;  
2) import java.util.stream.*;  
3) public class Test  
4) {  
5)     public static void main(String[] args)  
6)     {  
7)         Map<String,String> m=new HashMap<>();  
8)         m.put("A","Apple");  
9)         m.put("B","Banana");  
10)        m.put("C",null);  
11)        m.put("D","Dog");  
12)        m.put("E",null);  
13)        List<String> l=m.entrySet().stream().map(e->e.getKey()).collect(Collectors.toList());  
14)        System.out.println(l);  
15)  
16)        List<String> l2=m.entrySet().stream()  
17)            .flatMap(e->Stream.ofNullable(e.getValue())).collect(Collectors.toList());  
18)        System.out.println(l2);  
19)  
20)    }  
21) }
```

Output:

[A, B, C, D, E]
[Apple, Banana, Dog]



The Java Shell (RPEL)

JShell Agenda

1) Introduction to the JShell	81
2) Getting Started with JShell	82
3) Getting Help from the JShell	86
4) Understanding JShell Snippets	95
5) Editing and Navigating Code Snippets	100
6) Working with JShell Variables	102
7) Working with JShell Methods	107
8) Using An External Editor with JShell	113
9) Using classes, interfaces and enum with JShell	115
10) Loading and Saving Snippets in JShell	117
11) Using Jar Files in the JShell	121
12) How to customize JShell Startup	123
13) Shortcuts and Auto-Completion of Commands	127



UNIT 1: Introduction to the JShell

Jshell is also known as interactive console.

JShell is Java's own REPL Tool.

REPL means → Read, Evaluate, Print and Loop

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

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

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

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

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

Python → IDLE

Apple's Swift Programming Language → Playground

Limitations of JShell:

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

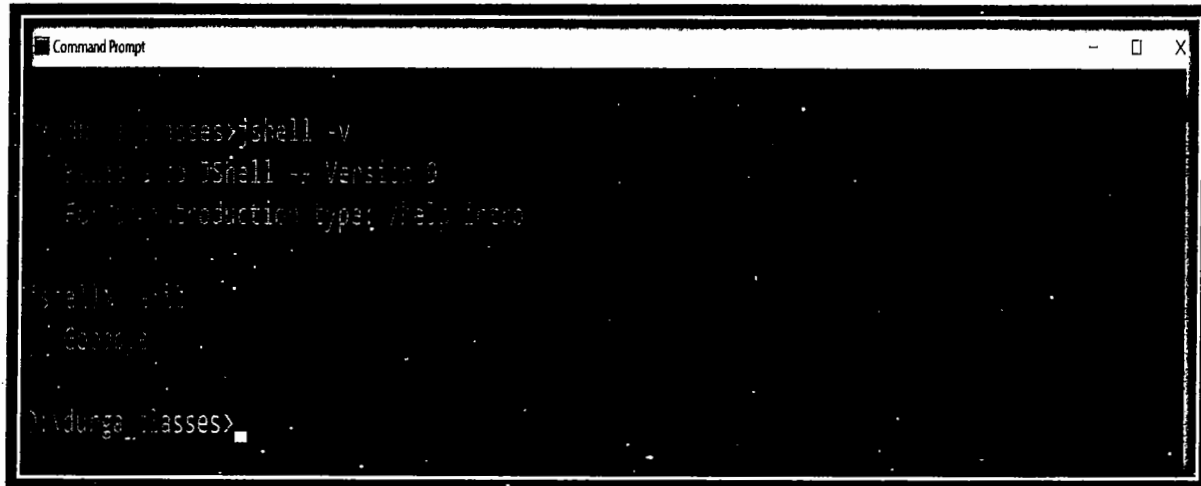


UNIT-2: Getting Started with JShell

Starting and Stopping JShell:

Open the jshell from the command prompt in verbose mode

```
jshell -v
```



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

jshell>/exit
Goodbye

D:\durga_classes>
```

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

How to exit jshell:

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

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

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

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

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

```
jshell> 20-30*6/2
$2 ==> -70
```



```
| created scratch variable $2 : int
jshell> System.out.println("DURGASOFT")
DURGASOFT
```

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

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

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

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

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

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

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

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

Can you check whether the following will work or not?

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

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

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



```
| import Java.util.regex.*
| import Java.util.stream.*
jshell> ArrayList<String> l=new ArrayList<String>();
l ==> []

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

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

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

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

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

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

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

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

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

```
jshell> System.out.println(x+y)
| Error:
| cannot find symbol
| symbol: variable x
| System.out.println(x, y)
|      ^
| Error:
| cannot find symbol
| symbol: variable y
| System.out.println(x+y)
```



```
jshell> Sytsem.out.println("Durga")
```

```
| Error:  
| package Sytsem does not exist  
| Sytsem.out.println("Durga")  
| ^_____  
|
```

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

Eg:

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

```
D:\durga_classes>Javac Test.java
```

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

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

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

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

Conclusions:

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



UNIT - 3: Getting Help from the JShell

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

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

Agenda:

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

Type jshell --help from normal command prompt

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

Type jshell --version from normal command prompt

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

jshell> /help intro

4) **For List of commands:**

type /help from jshell

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

jshell>/help commandname

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

just type / followed by tab

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

jshell>/command - tab

jshell> /list -

-all -history -start



1) To know list of options allowed with jshell

Type `jshell --help` from normal command prompt

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

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

where possible options include:

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

2) To know the version of jshell

Type `jshell --version` from normal command prompt

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




3) To know introduction of jshell

```
jshell> /help intro
```

```
| intro
```

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

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

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

```
| or a Java statement or import.
```

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

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

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

```
| For a list of commands: /help
```

4) For List of commands

```
type /help from jshell
```

```
jshell> /help
```

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

```
| Or type one of the following commands:
```

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

```
| list the source you have typed
```

```
| /edit <name or id>
```

```
| edit a source entry referenced by name or id
```

```
| /drop <name or id>
```

```
| delete a source entry referenced by name or id
```

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

```
| Save snippet source to a file.
```

```
| /open <file>
```

```
| open a file as source input
```

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

```
| list the declared variables and their values
```

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

```
| list the declared methods and their signatures
```

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

```
| list the declared types
```

```
| /imports
```

```
| list the imported items
```

```
| /exit
```

```
| exit jshell
```



```
| /env [-class-path <path>] [-module-path <path>] [-add-modules <modules>] ...  
|     view or change the evaluation context  
| /reset [-class-path <path>] [-module-path <path>] [-add-modules <modules>]...  
|     reset jshell  
| /reload [-restore] [-quiet] [-class-path <path>] [-module-path <path>]...  
|     reset and replay relevant history -- current or previous (-restore)  
| /history  
|     history of what you have typed  
| /help [<command>|<subject>]  
|     get information about jshell  
| /set editor|start|feedback|mode|prompt|truncation|format ...  
|     set jshell configuration information  
| /? [<command>|<subject>]  
|     get information about jshell  
| /!  
|     re-run last snippet  
| /<id>  
|     re-run snippet by id  
| /-<n>  
|     re-run n-th previous snippet
```

| For more information type '/help' followed by the name of a
| command or a subject.

| For example '/help /list' or '/help intro'.

| Subjects:

| intro

| an introduction to the jshell tool

| shortcuts

| a description of keystrokes for snippet and command completion,
| information access, and automatic code generation

| context

| the evaluation context options for /env /reload and /reset

5) To get information about a particular command

jshell>/help commandname

jshell> /help list

| /list

| Show the source of snippets, prefaced with the snippet id.

| /list

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



| /list -start

| List the automatically evaluated start-up snippets

| /list -all

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

| /list <name>

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

| /list <id>

| List the snippet with the specified snippet id

To get Information about methods command

jshell> /help methods

| /methods

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

| /methods

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

| /methods <name>

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

| /methods <id>

| List the jshell method with the specified snippet id

| /methods -start

| List the automatically added start-up jshell methods

| /methods -all

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

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

just type / followed by tab

jshell> /

!/ /? /drop /edit /env /exit /help

/history /imports /list /methods /open /reload /reset

/save /set /types /var

<press tab again to see synopsis>



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

jshell> /
/!
re-run last snippet

/-<n>
re-run n-th previous snippet

/**<id>**
re-run snippet by id

/?
get information about jshell

/drop
delete a source entry referenced by name or id

/edit
edit a source entry referenced by name or id

/env
view or change the evaluation context

/exit
exit jshell

/help
get information about jshell

/history
history of what you have typed

/imports
list the imported items

/list
list the source you have typed

/methods
list the declared methods and their signatures

/open
open a file as source input

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



`/reset`
reset jshell

`/save`
Save snippet source to a file.

`/set`
set jshell configuration information

`/types`
list the declared types

`/vars`
list the declared variables and their values

<press tab again to see full documentation>

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

jshell> /
/!
Reevaluate the most recently entered snippet.

<press tab to see next command>

jshell> /
/-<n>
Reevaluate the n-th most recently entered snippet.

<press tab to see next command>

jshell> /
/<id>
Reevaluate the snippet specified by the id.

<press tab to see next command>

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

jshell>/command - tab

jshell> /list -
-all -history -start

<press tab again to see synopsis>

jshell> /list -



If we press tab again then we will get synopsis:

jshell> /list -
list the source you have typed

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

If we press tab again then we will get documentation:

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

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

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

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

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

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



Conclusions:

1) To know list of options allowed with jshell

Type `jshell --help` from normal command prompt

2) To know the version of jshell

Type `jshell --version` from normal command prompt

3) To know introduction of jshell

`jshell> /help intro`

4) For List of commands

type `/help` from jshell

5) To get information about a particular command

`jshell>/help commandname`

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

just type `/` followed by tab

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

`jshell>/command - tab`

`jshell> /list -`

`-all -history -start`



UNIT - 4: Understanding JShell Snippets

What are Coding Snippets?

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

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

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

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

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

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

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

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

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

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




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

```
jshell> /list
```

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

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

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

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

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

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

All these are default imports to the jshell.

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

```
jshell> /list -all
```

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



```
s7 : import Java.util.function.*;
s8 : import Java.util.prefs.*;
s9 : import Java.util.regex.*;
s10 : import Java.util.stream.*;
1 : System.out.println("Hello")
2 : int x=10;
3 : 10+20
4 : $3>x
e1 : String s =10;
5 : String s= "Durga";
6 : public void m1()
{
    System.out.println("hello");
}
7 : m1()
```

We can access snippets by using id directly.

```
jshell> /list 1
```

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

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

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

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

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

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

```
jshell> /list m1
```

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

```
jshell> /list x
```

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

```
jshell> /list s
```



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

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

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

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

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

```
jshell> /list

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

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

```
jshell> /list

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



```
jshell> /4
$3>x
| Error:
| cannot find symbol
| symbol: variable $3
| $3>x
| ^^
```

Conclusions:

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

```
jshell>/list
```

2. In the future we can access the snippet with id directly without retyping whole snippet.

```
jshell>/list id
```

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

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

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

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

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

```
jshell> /list -all
```

6. We can access snippets by using id directly.

```
jshell> /list 1
```

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

```
jshell> /list m1
```

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

```
jshell> /3
```

9. We can use drop command to drop a snippet (Making it inactive)

We can drop snippet by name or id.

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

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



UNIT – 5: Editing and Navigating Code Snippets

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

```
jshell> /list
```

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

```
jshell> /history
```

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

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



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



UNIT – 6: Working with JShell Variables

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

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

In JShell, there are 2 types of variables

1. Explicit variables
2. Implicit variables or Scratch variables

Explicit variables:

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

Eg:

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

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

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

Implicit Variables:

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

Eg:

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



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

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

Based on requirement we can use these scratch variables also.

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

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

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

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

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

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

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

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

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

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




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

To List out All Active variables of JShell:

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

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

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

We can drop a variable by using /drop command

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

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



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

We can create complex variables also

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

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

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

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

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

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

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

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

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

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



Hello:Durga

\$11 ==> Java.io.PrintStream@10bdf5e5

| created scratch variable \$11 : PrintStream

jshell> \$11.printf("Hello")

Hello\$12 ==> Java.io.PrintStream@10bdf5e5

| created scratch variable \$12 : PrintStream

jshell> /vars

| String s = "Durga"

| boolean \$4 = true

| String x = "DURGASOFT"

| String s1 = "Hello"

| List<String> heroes = [Ameer, Sharukh, Salman]

| List<String> heroines = [Katrina, Kareena, Deepika]

| List<List<String>> l = [[Ameer, Sharukh, Salman],

[Katrina, Kareena, Deepika]]

| PrintStream \$11 = Java.io.PrintStream@10bdf5e5

| PrintStream \$12 = Java.io.PrintStream@10bdf5e5

FAQs:

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



UNIT – 7: Working with JShell Methods

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

Eg:

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

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

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

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

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

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

| created method m1()

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

| created method m1(int)

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

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

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



```
| /methods
|   List the name, parameter types, and return type of the current active jshell methods
|
| /methods <name>
|   List jshell methods with the specified name (preference for active methods)
|
| /methods <id>
|   List the jshell method with the specified snippet id
|
| /methods -start
|   List the automatically added start-up jshell methods
|
| /methods -all
|   List all snippets including failed, overwritten, dropped, and start-up
```

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

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

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

```
jshell> public int m1(int i){return 10;}
| replaced method m1(int)
| update overwrote method m1(int)
```

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

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

In the JShell we can create more complex methods also.

Eg1: To print the number of occurrences of specified character i.e. the given String

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



```
5)      {
6)          if(s.charAt(i)==ch)
7)          {
8)              count++;
9)          }
10)     }
11)     System.out.println("The number of occurrences:"+count);
12) }
```

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

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

Eg 2: To print the sum of given integers

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

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

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

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

Eg1: Usage of undeclared variable inside method body

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

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

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



```
jshell> int x=10
x ==> 10
| created variable x : int
| update modified method m1()
```

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

Eg 2: Usage of undeclared method inside method body

```
jshell> public void m1()
...> {
...> m2();
...> }
| created method m1(), however, it cannot be invoked until method m2() is declared
```

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

```
jshell> public void m2()
...> {
...> System.out.println("Hello DURGASOFT");
...> }
| created method m2()
| update modified method m1()
```

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

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

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

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

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

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

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



```
jshell> /methods
```

```
| void m1()  
| void m1(int)  
| void m2()  
| void m3()
```

```
jshell> /drop m3
```

```
| dropped method m3()
```

```
jshell> /methods
```

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

```
jshell> /drop m1
```

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

```
| Use one of:
```

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

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

```
jshell> /methods
```

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

```
jshell> /list
```

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

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

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

```
jshell> /drop 2
```

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

```
jshell> /methods
```

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




FAQs:

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



UNIT – 8: Using An External Editor with JShell

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

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

```
jshell> /edit
```

diagram(image) of inbuilt-editor

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

Eg:

```
jshell> /set editor "C:\\WINDOWS\\system32\\notepad.exe"
jshell> /set editor "C:\\Program Files\\EditPlus\\editplus.exe"
```

How to Set Notepad as editor to JShell:

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

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

```
jshell> /edit
```

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

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

```
jshell> /set editor -retain
| Editor setting retained: C:\\WINDOWS\\system32\\notepad.exe
```

How to set EditPlus as editor to JShell:

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

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



How to set default editor once again:

We have to type the following command from the jshell

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

To make default editor as permanent:

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

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

FAQs:

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



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

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

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

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

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

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

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

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

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

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

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

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



| created variable s : Student

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

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

| created scratch variable \$3 : String

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

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

| created scratch variable \$4 : int

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

```
jshell> /edit
```

| created interface Interf

| created enum Beer

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

```
interface static method
```

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

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

| created scratch variable \$8 : String



UNIT – 10: Loading and Saving Snippets in JShell

We can load and save snippets from the file.

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

`mynippets.jsh`:

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

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

```
jshell> /list
```

```
jshell> /open mynippets.jsh
```

```
jshell> /list
```

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

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

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

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

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



Saving JShell snippets to the file:

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

```
jshell> /help save
```

```
| /save <file>
```

```
| Save the source of current active snippets to the file.
```

```
| /save -all <file>
```

```
| Save the source of all snippets to the file.
```

```
| Includes source including overwritten, failed, and start-up code.
```

```
| /save -history <file>
```

```
| Save the sequential history of all commands and snippets entered since jshell was launched.
```

```
| /save -start <file>
```

```
| Save the current start-up definitions to the file.
```

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

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

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

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

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

```
jshell> /ex
```

```
| Goodbye
```

```
D:\>type active.jsh
```

```
int x=10;
```

```
x
```

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

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

```
D:\>type start.jsh
```

```
import Java.io.*;
```

```
import Java.math.*;
```

```
import Java.net.*;
```

```
import Java.nio.file.*;
```

```
import Java.util.*;
```

```
import Java.util.concurrent.*;
```

```
import Java.util.function.*;
```

```
import Java.util.prefs.*;
```



```
import Java.util.regex.*;  
import Java.util.stream.*;
```

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

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

How to Reload Previous state (session) of JShell:

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

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

Eg:

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

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

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

```
jshell> /list
```

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

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

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

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




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

How to reset JShell State:

We can reset JShell state by using /reset command.

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

Eg:

```
jshell> /list

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

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

```
jshell> /list
```



UNIT – 11: Using Jar Files in the JShell

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

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

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

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

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

Demo Program to get all employees information from oracle database:

mynippets.jsh:

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

DaTabase info:

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

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

```
jshell> /open mysnippets.jsh
jshell> getEmplInfo()
```



100..Sunny..1000.0..Mumbai
200..Bunny..2000.0..Hyd
300..Chinny..3000.0..Hyd
400..Vinny..4000.0..Delhi

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

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

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

```
jshell> /open mysnipets.jsh
```

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

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



UNIT – 12: How to customize JShell Startup

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

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

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

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

mystartup.jsh:

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

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

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

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

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

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

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

```
jshell> /list
```



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

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

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

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

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

```
jshell> /list
```

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

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

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

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

```
jshell> /list
```

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

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



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

Q. What is the difference between the following?

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

Need of PRINTING Option at the startup:

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

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

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

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

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



```
s19 : void println(Object obj) { System.out.println(obj); }  
s20 : void printf(Locale l, String format, Object... args) { System.out.printf(l, format, args);  
}  
s21 : void printf(String format, Object... args) { System.out.printf(format, args); }
```

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

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

Note:

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

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

Note:

Various allowed options with `--startup` are :

1. DEFAULT
2. JAVASE
3. PRINTING



UNIT – 13: Shortcuts and Auto-Completion of Commands

Shortcut for Creating Variables:

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

```
jshell> "Durga" // just press "Shift+Tab followed by v"
```

```
jshell> String s= "Durga"
```

We have to provide only name s

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

```
jshell> double d = 10.5
```

Shortcut for auto-import:

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

```
jshell> Connection // press "Shift+Tab followed by i"
```

```
0: Do nothing
```

```
1: import: com.sun.jdi.connect.spi.Connection
```

```
2: import: Java.sql.Connection
```

```
Choice: //enter 2
```

```
Imported: Java.sql.Connection
```

```
jshell> /imports
```

```
| import Java.io.*
```

```
| import Java.math.*
```

```
| import Java.net.*
```

```
| import Java.nio.file.*
```

```
| import Java.util.*
```

```
| import Java.util.concurrent.*
```

```
| import Java.util.function.*
```

```
| import Java.util.prefs.*
```

```
| import Java.util.regex.*
```

```
| import Java.util.stream.*
```

```
| import Java.sql.Connection
```




Auto Completion commands :

1. To get all static members of the class:

```
jshell>classname.<Tab>
```

Eg:

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

2. To get all instance members of class:

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

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

3. To get signature and documentation of a method:

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

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

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

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

Signatures:

String String.substring(int beginIndex)

String String.substring(int beginIndex, int endIndex)

<press Tab again to see documentation>



jshell> s.substring(<Tab>

String String.substring(int beginIndex)

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

Examples:

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

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

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

Parameters:

beginIndex - the beginning index, inclusive.

Returns:

the specified substring.

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

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

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

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



The Java Platform Module System (JPMS)

JPMS Agenda

- 1) Introduction
- 2) What is the need of JPMS?
- 3) Jar Hell or Classpath Hell
- 4) What is a Module
- 5) Steps to Develop First Module Based Application
- 6) Various Possible Ways to Compile & run a Module
- 7) Inter Module Dependencies
- 8) requires and exports directive
- 9) JPMS vs NoClassDefFoundError
- 10) Transitive Dependencies (requires with transitive Keyword)
- 11) Optional Dependencies (Requires Directive with static keyword)
- 12) Cyclic Dependencies
- 13) Qualified Exports
- 14) Module Graph
- 15) Observable Modules
- 16) Aggregator Module
- 17) Package Naming Conflicts
- 18) Module Resolution Process (MRP)



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 place 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.

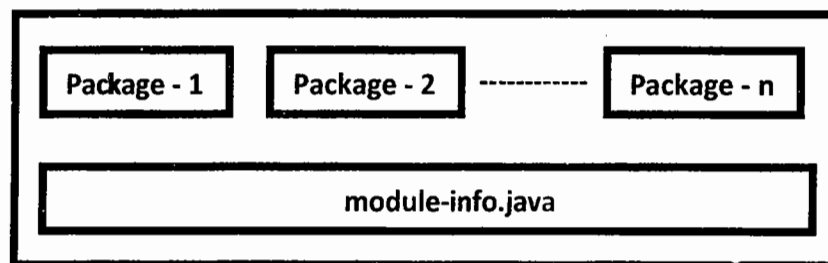
Hence jar file is nothing but a group of packages and each package contains several .class files.

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.



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

- java.base
- java.logging
- java.sql
- java.desktop(AWT/Swing)
- java.rmi etc



java.base module acts as base for all java 9 modules.
We can find module of a class by using `getModule()` method.

Eg: `System.out.println(String.class.getModule());`//module java.base

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.

Demo Program to demonstrate *NoClassDefFoundError*:

```
durgajava9
|-A1.java
|-A2.java
|-A3.java
|-Test.java
```

A1.java:

```
1) package pack1;
2) public class A1
3) {
4)     public void m1()
5)     {
6)         System.out.println("pack1.A");
7)     }
8) }
```

A2.java

```
1) package pack2;
2) import pack1.A1;
3) public class A2
4) {
5)     public void m2()
6)     {
7)         System.out.println("pack2.A2 method");
8)         A1 a = new A1();
9)         a.m1();
10)    }
11) }
```



A3.java:

```
1) import pack2.A2;
2) class Test
3) {
4)     public static void main(String[] args)
5)     {
6)         System.out.println("Test class main");
7)         A2 a= new A2();
8)         a.m2();
9)     }
10)}
```

```
D:\durgajava9>javac -d . A1.java
D:\durgajava9>javac -d . A2.java
D:\durgajava9>javac -d . A3.java
D:\durgajava9>javac Test.java
```

```
durgajava9
|-Test.class
|-pack1
|-A1.class
|-pack2
|-A2.class
```

At runtime, by mistake if pack1 is not available then after executing some part of the code in the middle, we will get `NoClassDefFoundError`.

```
D:\durgajava9>java Test
Test class main
pack2.A2 method
Exception in thread "main" java.lang.NoClassDefFoundError: pack1/A1
```

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.

classpath = jar1;jar2;jar3;jar4

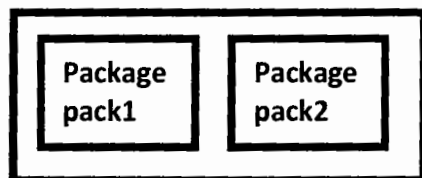
If jar4 requires Test.class file of jar3. But Different versions of Test.class is available in jar1, jar2 and jar3. In this case jar1 Test.class file will be considered, because JVM will always search from Left to Right in the classpath. It will create version conflicts and causes abnormal behavior of program.



But in java9 module system, there is a way to specify dependent modules information for every module separately. 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.



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.

Module can offer Strong Encapsulation than Jar File.

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 modules.

Q. Explain differences between jar file and Java 9 module

Jar	Module
1) Jar is a group of packages and each package contains several classes	1) 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	2) For every module we have to maintain a special file module-info.java to specify module dependencies
3) 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.	3) 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 its execution. Hence there is no chance of NoClassDefFoundError in the middle of execution.
4) 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 file then there may be a chance of Version conflicts and results abnormal behavior of our application	4) 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.
5) 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 any component from the jar file. Hence there may be a chance of security problems	5) In module there is a mechanism to control access to the packages. Only exported packages are visible to other modules. Hence there is no chance of security problems
6) Jars follows monolithic structure and applications will become heavy weight and not suitable for small devices.	6) Modules follow distributed structure and applications will become light weighted and suitable for small devices.
7) Jar files approach cannot be used for IOT devices and micro services.	7) Modules based approach can be used for IOT devices and micro services



Q. What is Jar Hell or Classpath Hell?

The problems with use of jar files are:

1. *NoClassDefFoundError* in the middle of program execution
2. Version Conflicts and Abnormal behavior of program
3. Lack of Security
4. Bigger Size

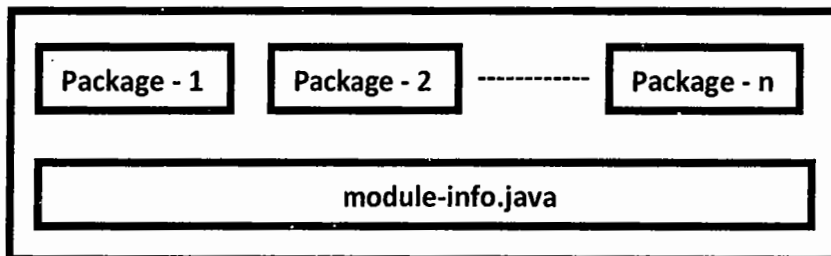
This set of Problems is called Jar Hell OR Classpath Hell. To overcome this, we should go for JPMS.

Q. What are various Goals/Benefits of JPMS?

1. Reliable Configuration
 2. Strong Encapsulation & Security
 3. Scalable Java Platform
 4. Performance and Memory Improvements
- etc

What is a Module:

Module is nothing but collection of packages. Each module should compulsory contains a special configuration file: `module-info.java`.



We can define module dependencies inside `module-info.java` file.

```
module moduleName
```

```
{
```

Here we have to define module dependencies which represents

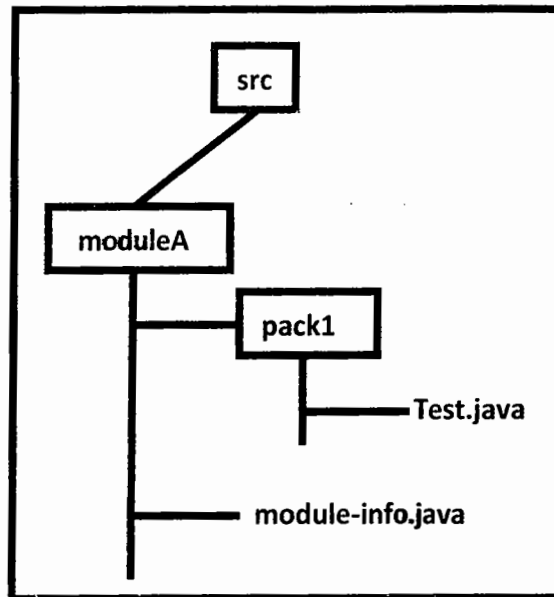
1. What other modules required by this module?
2. What packages exported by this module for other modules?

etc

```
}
```



Steps to Develop First Module Based Application:



Step-1: Create a package with our required classes

```
1) package pack1;
2) public class Test
3) {
4)     public static void main(String[] args)
5)     {
6)         System.out.println("First Module in JPMS");
7)     }
8) }
```

Step-2: Writing module-info.java

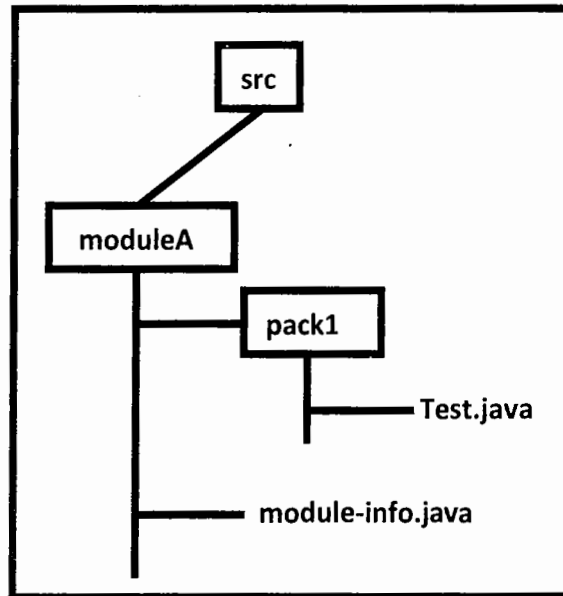
For every module we should write a special file named with module-info.java. In this file we have to define dependencies of module.

```
module moduleA
{
}
```



Step-3: Arrange all files in the required package structure

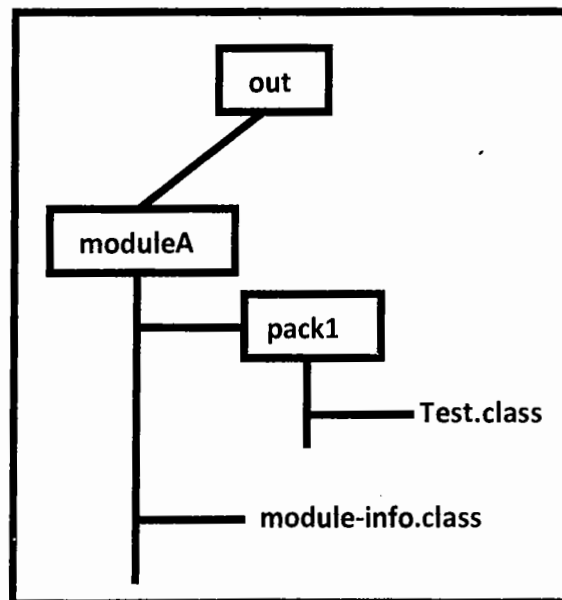
Arrange all the files according to required folder structure



Step-4: Compile module with --module-source-path option

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

The generated class file structure is:





Step-5: Run the class with --module-path option

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

Output: First Module in JPMS

Case-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
```

Case-2:

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
In the above application inside Test.java if we comment package statement

```
//package pack1;
```

```
1) public class Test
2) {
3)     public static void main(String[] args)
4)     {
5)         System.out.println("First Module in JPMS");
6)     }
7) }
```

error: unnamed package is not allowed in named modules

Case-3:

The module name should not ends with digit(like module1,module2 etc),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 Possible Ways to Compile a Module:

```
javac --module-source-path src -d out -m moduleA
javac --module-source-path src -d out --module moduleA
javac --module-source-path src -d out
    src/moduleA/module-info.java src/moduleA/pack1/Test.java
javac --module-source-path src -d out
    C:/Users/Durga/Desktop/src/moduleA/module-info.java
C:/Users/Durga/Desktop/src/moduleA/pack1/Test.java
```

Various Possible Ways to Run a Module:

```
java --module-path out --add-modules moduleA pack1.Test
java --module-path out -m moduleA/pack1.Test
java --module-path out --module moduleA/pack1.Test
```

Inter Module Dependencies:

Within the application we can create any number of modules and one module can use other modules.

We can define module dependencies inside module-info.java file.

```
module moduleName
{
    Here we have to define module dependencies which represents
    1. What other modules required by this module?
    2. What packages exported by this module for other modules?
    etc
}
```

Mainly we can use the following 2 types of directives

1. requires directive:

It can be used to specify the modules which are required by current module.

Eg:

```
1) module moduleA
2) {
3)     requires moduleB;
4) }
```

It indicates that moduleA requires members of moduleB.



Note:

1. We cannot use same requires directive for multiple modules. For every module we have to use separate requires directive.

requires moduleA,moduleB; → invalid

2. We can use requires directive only for modules but not for packages and classes.

2. exports directive:

It can be used to specify what packages exported by current module to the other modules.

Eg:

```
1) module moduleA
2) {
3)   exports pack1;
4) }
```

It indicates that moduleA exporting pack1 package so that this package can be used by other modules.

Note: We cannot use same exports directive for exporting multiple packages. For every package a separate exports directive must be required.

exports pack1,pack2; → invalid

Note: Be careful about syntax requires directive always expecting module name where as exports directive expecting package name.

```
1) module modulename
2) {
3)   requires modulename;
4)   exports packagename;
5) }
```

Note:

By default all packages present in a module are private to that module. If module exports any package only that particular package is accessible by other modules. Non exporting packages cannot be accessed by other modules.

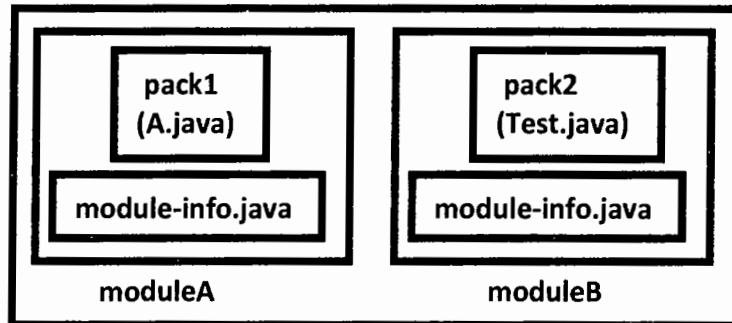
Eg: Assume moduleA contains 2 packages pack1 and pack2. If moduleA exports only pack1 then other modules can use only pack1. pack2 is just for its internal purpose and cannot be accessed by other modules.



```
1) module moduleA
2) {
3)   exports pack1;
4) }
```

Demo program for inter module dependencies:

Rectangle diagram which represents total application



moduleA components:

A.java:

```
1) package pack1;
2) public class A
3) {
4)   public void m1()
5)   {
6)     System.out.println("Method of moduleA");
7)   }
8) }
```

module-info.java:

```
1) module moduleA
2) {
3)   exports pack1;
4) }
```



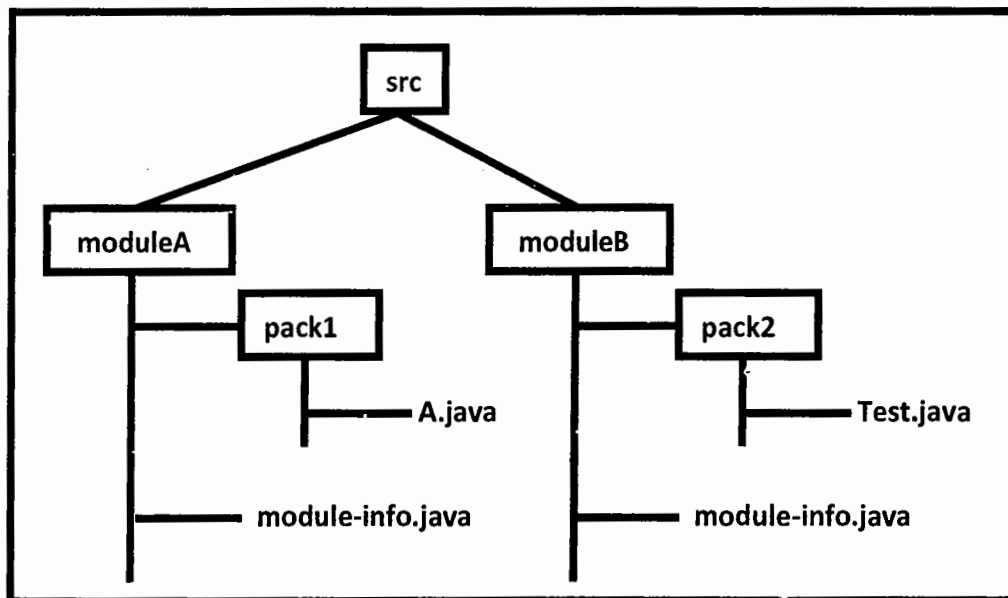
moduleB components:

Test.java:

```
1) package pack2;  
2) import pack1.A;  
3) public class Test  
4) {  
5)     public static void main(String[] args)  
6)     {  
7)         System.out.println("moduleB accessing members of moduleA");  
8)         A a = new A();  
9)         a.m1();  
10)    }  
11) }
```

module-info.java:

```
1) module moduleB  
2) {  
3)     requires moduleA;  
4) }
```

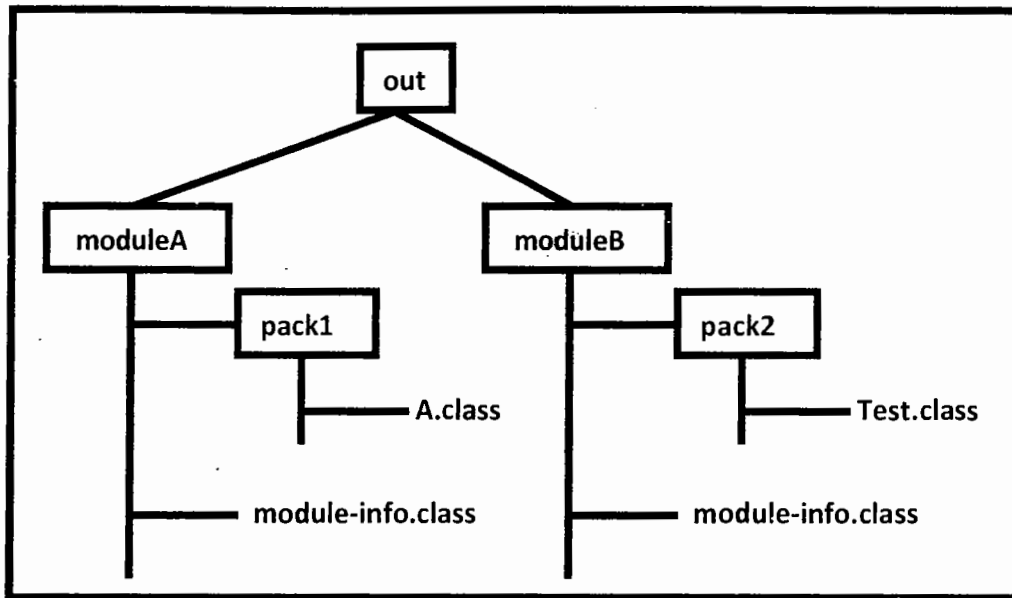


Compilation:

C:\Users\Durga\Desktop>javac --module-source-path src -d out -m moduleA,moduleB

Note: space is not allowed between the module names otherwise we will get error.

C:\Users\Durga\Desktop>javac --module-source-path src -d out -m moduleA, moduleB
error: Class names, 'moduleP' are only accepted if annotation processing is explicitly requested



Execution:

```
C:\Users\Durga\Desktop>java --module-path out -m moduleB/pack2.Test
```

Output:

moduleB accessing members of moduleA
Method of moduleA

Case-1:

Even though class A is public, if moduleA won't export pack1, then moduleB cannot access A class.

Eg:

```
1) module moduleA
2) {
3)    //exports pack1;
4) }
```

```
C:\Users\Durga\Desktop>javac --module-source-path src -d out -m moduleA,moduleB
src\moduleB\pack2\Test.java:2: error: package pack1 is not visible
import pack1.A;
    ^
```

(package pack1 is declared in module moduleA, which does not export it)

1 error

Case-2:

We have to export only packages. If we are trying to export modules or classes then we will get compile time error.



Eg-1: exporting module instead of package

```
1) module moduleA
2) {
3)     exports moduleA;
4) }
```

C:\Users\Durga\Desktop>javac --module-source-path src -d out -m moduleA, moduleB
src\moduleB\pack2\Test.java:2: error: package pack1 is not visible
import pack1.A;

^

(package pack1 is declared in module moduleA, which does not export it)
src\moduleA\module-info.java:3: error: package is empty or does not exist: moduleA
exports moduleA;

In this case compiler considers moduleA as package and it is trying to search for that package.

Eg-2: exporting class instead of package:

```
1) module moduleA
2) {
3)     exports pack1.A;
4) }
```

C:\Users\Durga\Desktop>javac --module-source-path src -d out -m moduleA, moduleB
src\moduleB\pack2\Test.java:2: error: package pack1 is not visible
import pack1.A;

^

(package pack1 is declared in module moduleA, which does not export it)
src\moduleA\module-info.java:3: error: package is empty or does not exist: pack1.A
exports pack1.A;

^

2 errors

In this case compiler considers pack1.A as package and it is trying to search for that package.

Case-3:

If moduleB won't use "requires moduleA" directive then moduleB is not allowed to use members of moduleA, even though moduleA exports.

```
1) module moduleB
2) {
3)     //requires moduleA;
4) }
```

C:\Users\Durga\Desktop>javac --module-source-path src -d out -m moduleA, moduleB
src\moduleB\pack2\Test.java:2: error: package pack1 is not visible
import pack1.A;



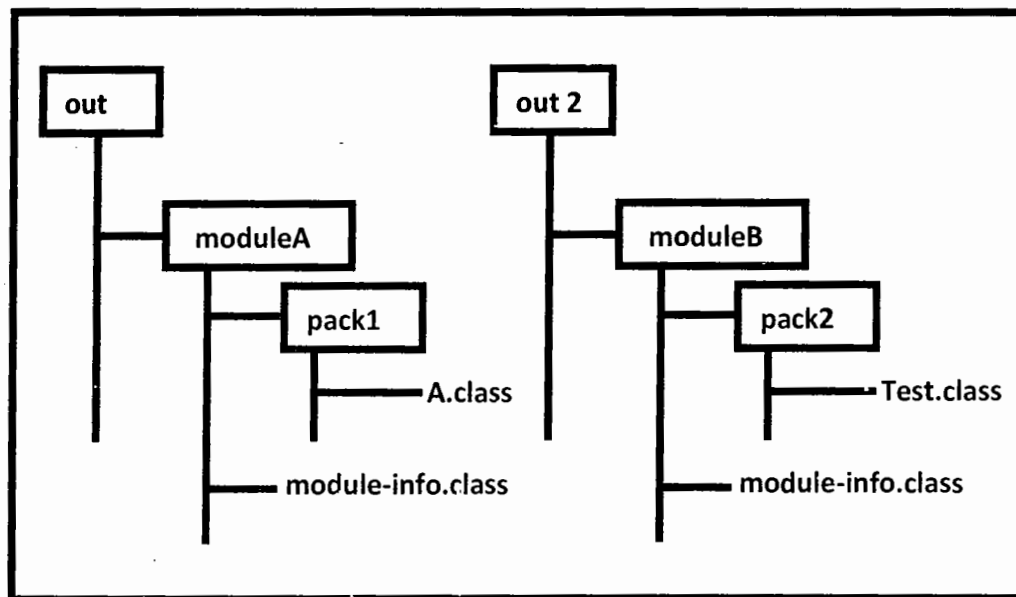
^
(package pack1 is declared in module moduleA, but module moduleB does not read it)
1 error

Case-4:

If compiled codes are available in different packages then how to run?

We have to use special option: `--upgrade-module-path`

If compiled codes of moduleA is available in out and compiled codes of moduleB available in out2



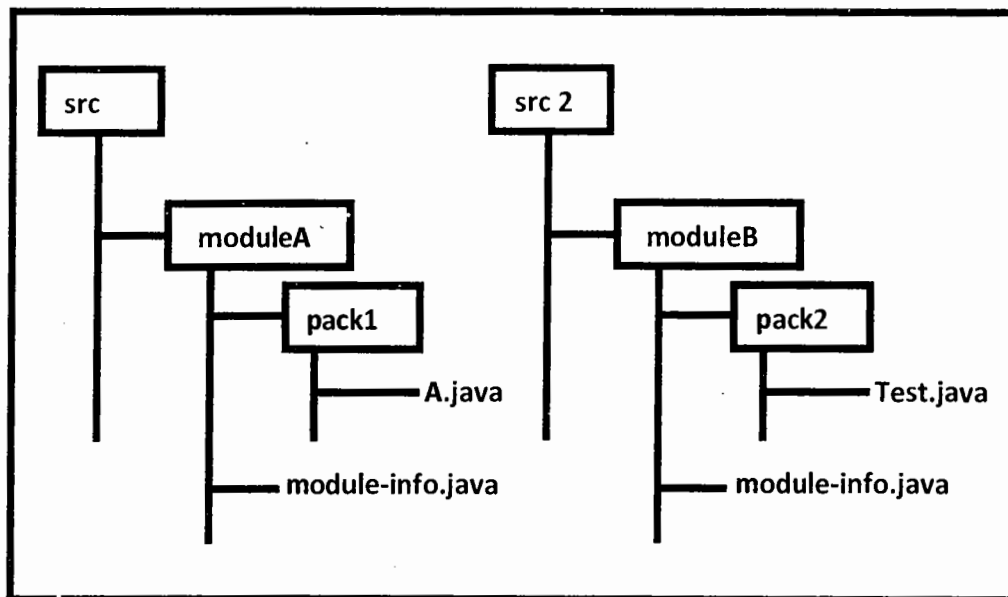
C:\Users\Durga\Desktop>java --upgrade-module-path out;out1 -m moduleB/pack2.Test
moduleB accessing members of moduleA
Method of moduleA



Case-5:

If source codes of two modules are in different directories then how to compile?

Assume moduleA source code is available in src directory and moduleB source code is available in src2



```
C:\Users\Durga\Desktop>javac --module-source-path src;src2 -d out -m moduleA, moduleB
```

Q. Which of the following are meaningful?

- 1) `module moduleName`
- 2) {
- 3) 1. `requires modulename;`
- 4) 2. `requires modulename.packagename;`
- 5) 3. `requires modulename.packagename.classname;`
- 6) 4. `exports modulename;`
- 7) 5. `exports packagename;`
- 8) 6. `exports packagename.classname;`
- 9) }

Answer: 1 & 5 are Valid

Note: We can use exports directive only for packages but not modules and classes, and we can use requires directive only for modules but not for packages and classes.

Note: To access members of one module in other module, compulsory we have to take care the following 3 things.

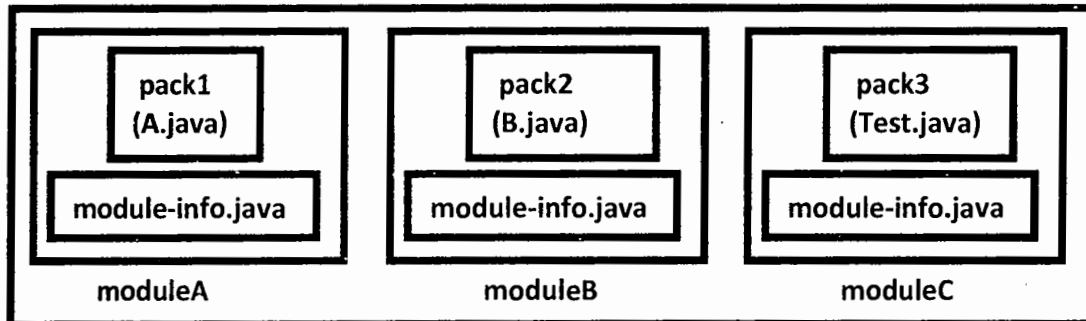
1. The module which is accessing must have requires dependency
2. The module which is providing functionality must have exports dependency
3. The member must be public.



JPMS vs NoClassDefFoundError:

In Java 9 Platform Modular System, JVM will check all dependencies at the beginning only. If any dependent module is missing then JVM won't start its execution. Hence there is no chance of NoClassDefFoundError in the middle of Program execution.

Demo Program:



Components of moduleA:

A.java:

```
1) package pack1;
2) public class A
3) {
4)     public void m1()
5)     {
6)         System.out.println("Method of moduleA");
7)     }
8) }
```

module-info.java:

```
1) module moduleA
2) {
3)     exports pack1;
4) }
```



Components of moduleB:

B.java:

```
1) package pack2;
2) import pack1.A;
3) public class B
4) {
5)     public void m2()
6)     {
7)         System.out.println("Method of moduleB");
8)         A a = new A();
9)         a.m1();
10)    }
11)}
```

module-info.java:

```
1) module moduleB
2) {
3)     requires moduleA;
4)     exports pack2;
5) }
```

Components of moduleC:

Test.java:

```
1) package pack3;
2) import pack2.B;
3) public class Test
4) {
5)     public static void main(String[] args)
6)     {
7)         System.out.println("Test class main method");
8)         B b = new B();
9)         b.m2();
10)    }
11)}
```

module-info.java:

```
1) module moduleC
2) {
3)     requires moduleB;
4) }
```



Compilation and Execution:

```
C:\Users\Durga\Desktop>javac --module-source-path src -d out -m moduleA, moduleB, moduleC
```

```
C:\Users\Durga\Desktop>java --module-path out -m moduleC/pack3.Test
```

Test class main method

Method of moduleB

Method of moduleA

If we delete compiled code of module (inside out folder), then JVM will raise error at the beginning only and JVM won't start program execution.

```
C:\Users\Durga\Desktop>java --module-path out -m moduleC/pack3.Test
```

Error occurred during initialization of boot layer

java.lang.module. FindException: Module moduleA not found, required by moduleB

But in Non Modular programming, JVM will start execution and in the middle, it will raise NoClassDefFoundError.

Hence in Java Platform Module System, there is no chance of getting NoClassDefFoundError in the middle of program execution.

Transitive Dependencies (requires with transitive Keyword):

$A \rightarrow B, B \rightarrow C \implies A \rightarrow C$

This property in mathematics is called Transitive Property.

Student1 requires Material, only for himself, if any other person asking he won't share.

```
1) module student1
2) {
3)   requires material;
4) }
```

"Student1 requires material not only for himself, if any other person asking him, he will share it"

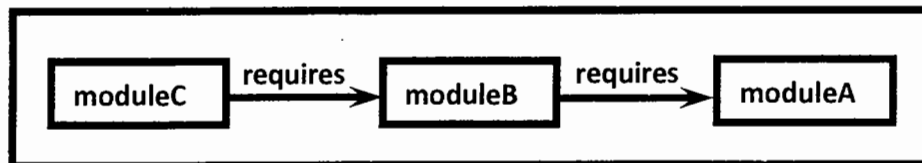
```
1) module Student1
2) {
3)   requires transitive material;
4) }
```

Sometimes module requires the components of some other module not only for itself and for the modules that requires that module also. For this requirement we can use transitive keyword.

The transitive keyword says that "Whatever I have will be given to a module that asks me."

**Case-1:**

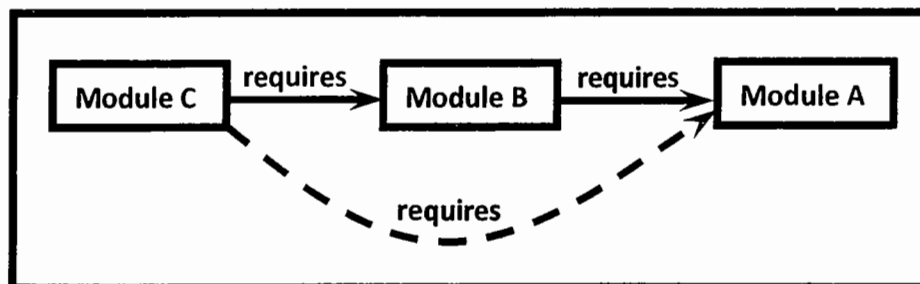
```
1) module moduleA
2) {
3)   exports pack1;
4) }
5) module moduleB
6) {
7)   requires moduleA;
8) }
9) module moduleC
10){
11)  requires moduleB;
12)}
```



In this case only moduleB is available to moduleC and moduleA is not available. Hence moduleC cannot use the members of moduleA directly.

Case-2:

```
1) module moduleA
2) {
3)   exports pack1;
4) }
5) module moduleB
6) {
7)   requires transitive moduleA;
8) }
9) module moduleC
10){
11)  requires moduleB;
12) }
```

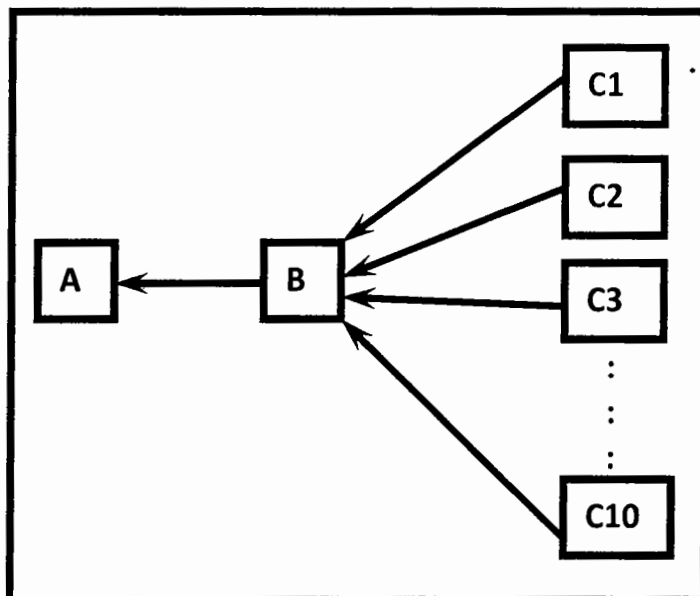


In this both moduleB and moduleA are available to moduleC. Now moduleC can use members of both modules directly.



Case Study:

Assume Modules C1, C2,...C10 requires Module B and Module B requires A.



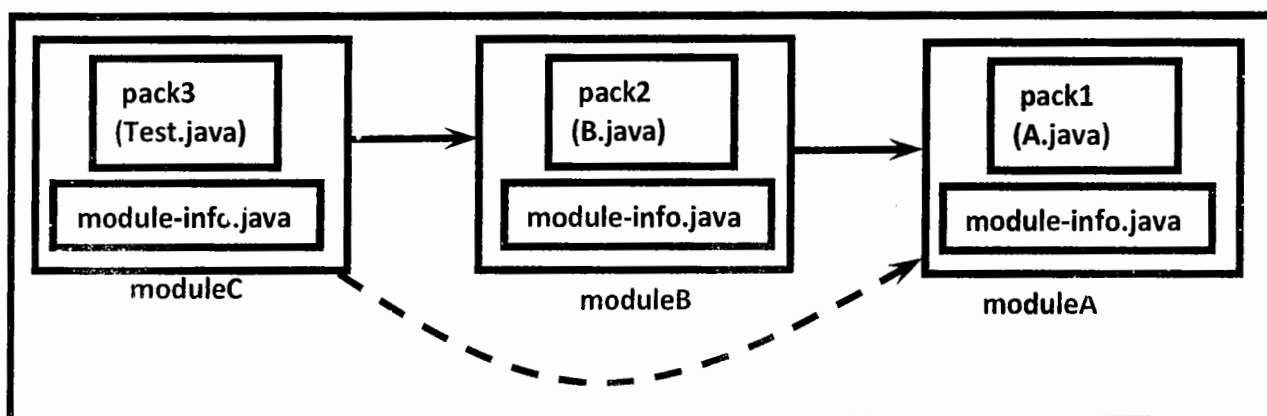
If we write "requires transitive A" inside module B

- 1) module B
- 2) {
- 3) **requires transitive A;**
- 4) }

Then module A is by default available to C1, C2,..., C10 automatically. Inside every module of C, we are not required to use "requires A" explicitly. Hence transitive keyword promotes code reusability.

Note: Transitive means implied readability i.e., Readability will be continues to the next level.

Demo Program for transitive keyword:





A.java:

```
1) package pack1;
2) public class A
3) {
4)     public void m1()
5)     {
6)         System.out.println("moduleA method");
7)     }
8) }
```

module-info.java:

```
1) module moduleA
2) {
3)     exports pack1;
4) }
```

moduleB components:

B.java:

```
1) package pack2;
2) import pack1.A;
3) public class B
4) {
5)     public A m2()
6)     {
7)         System.out.println("moduleB method");
8)         A a = new A();
9)         return a;
10)    }
11)}
```

module-info.java:

```
1) module moduleB
2) {
3)     requires transitive moduleA;
4)     exports pack2;
5) }
```



moduleC components:

Test.java:

```
1) package pack3;
2) import pack2.B;
3) public class Test
4) {
5)     public static void main(String[] args)
6)     {
7)         System.out.println("Test class main method");
8)         B b = new B();
9)         b.m2().m1();
10)    }
11)}
```

module-info.java:

```
1) module moduleC
2) {
3)     requires moduleB;
4) }
```

C:\Users\Durga\Desktop>javac --module-source-path src -d out -m moduleA, moduleB, moduleC

C:\Users\Durga\Desktop>java --module-path out -m moduleC/pack3.Test

Test class main method

moduleB method

moduleA method

In the above program if we are not using transitive keyword then we will get compile time error because moduleA is not available to moduleC.

javac --module-source-path src -d out -m moduleA,moduleB,moduleC

src\moduleC\pack3\Test.java:9: error: A.m1() in package pack1 is not accessible

b.m2().m1();

^

(package pack1 is declared in module moduleA, but module moduleC does not read it)

Optional Dependencies (Requires Directive with static keyword):

If Dependent Module should be available at compile time but optional at runtime, then such type of dependency is called Optional Dependency. We can specify optional dependency by using static keyword.

Syntax: requires static <modulename>



The static keyword is used to say that, "This dependency check is mandatory at compile time and optional at runtime."

Eg1:

```
1) module moduleB
2) {
3)   requires moduleA;
4) }
```

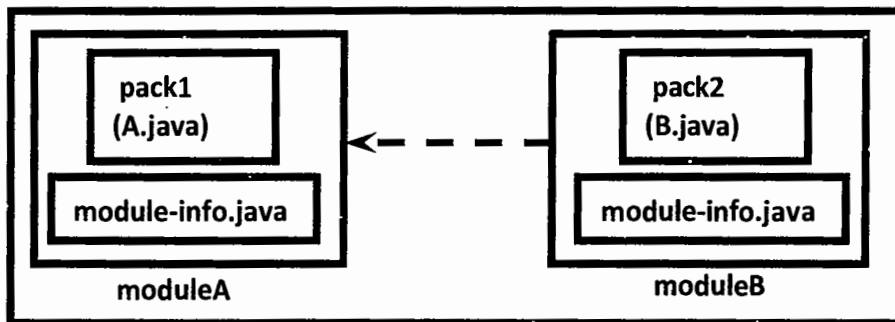
moduleA should be available at the time of compilation and runtime. It is not optional dependency.

Eg2:

```
1) module moduleB
2) {
3)   requires static moduleA;
4) }
```

At the time of compilation moduleA should be available, but at runtime it is optional. i.e., at runtime even moduleA is not available JVM will execute code.

Demo Program for Optional Dependency:



moduleA components:

A.java:

```
1) package pack1;
2) public class A
3) {
4)   public void m1()
5)   {
6)     System.out.println("moduleA method");
7)   }
8) }
```



module-info.java:

```
1) module moduleA
2) {
3)     exports pack1;
4) }
```

moduleB components:

Test.java:

```
1) package pack2;
2) public class Test
3) {
4)     public static void main(String[] args)
5)     {
6)         System.out.println("Optional Dependencies Demo!!!");
7)     }
8) }
9) }
```

module-info.java:

```
1) module moduleB
2) {
3)     requires static moduleA;
4) }
```

At the time of compilation both modules should be available. But at runtime, we can run moduleB Test class, even moduleA compiled classes are not available i.e., moduleB having optional dependency with moduleA.

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

```
C:\Users\Durga\Desktop>java --module-path out -m moduleB/pack2.Test
Optional Dependencies Demo!!!
```

If we remove static keyword and at runtime if we delete compiled classes of moduleA, then we will get error.

```
C:\Users\Durga\Desktop>java --module-path out -m moduleB/pack2.Test
Error occurred during initialization of boot layer
java.lang.module.FindException: Module moduleA not found, required by moduleB
```



Use cases of Optional Dependencies:

Usage of optional dependencies is very common in Programming world.

Sometimes we can develop library with optional dependencies.

Eg 1: If apache http Client is available use it, otherwise use HttpURLConnection.

Eg 2: If oracle module is available use it, otherwise use mysql module.

Why we should do this? For various reasons –

1. When distributing a library and we may not want to force a big dependency to the client.
2. On the other hand, a more advanced library may have performance benefits, so whatever module client needs, he can use.
3. We may want to allow easily pluggable implementations of some functionality. We may provide implementations using all of these, and pick the one whose dependency is found.

Q. What is the difference between the following?

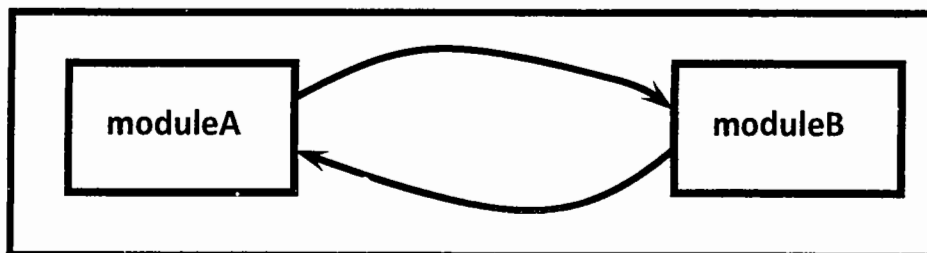
```
1) module moduleB
2) {
3)   1. requires moduleA;
4)   2. requires transitive moduleA
5)   3. requires static moduleA
6) }
```

Cyclic Dependencies:

If moduleA depends on moduleB and moduleB depends on moduleA, such type of dependency is called cyclic dependency.

Cyclic Dependencies between the modules are not allowed in java 9.

Demo Program:





moduleA components:

module-info.java

```
1) module moduleA
2) {
3)     requires moduleB;
4) }
```

moduleB components:

module-info.java

```
1) module moduleB
2) {
3)     requires moduleA;
4) }
```

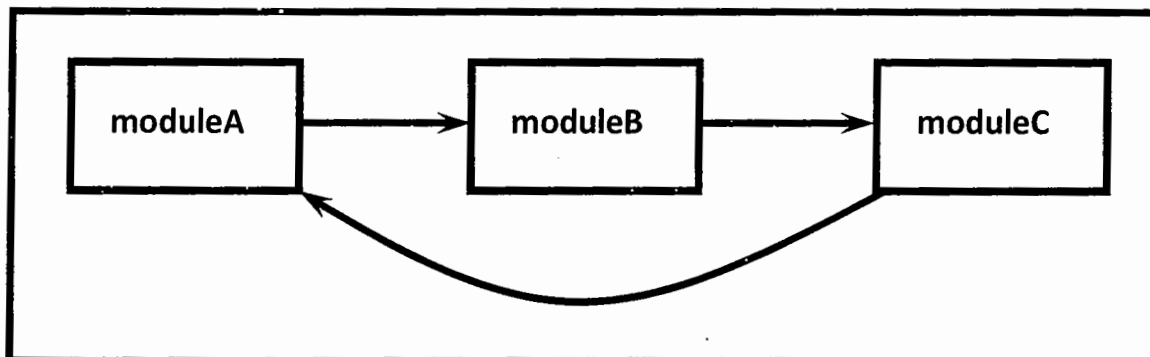
C:\Users\Durga\Desktop>javac --module-source-path src -d out -m moduleA, moduleB
src\moduleB\module-info.java:3: error: cyclic dependence involving moduleA
requires moduleA;

There may be a chance of cyclic dependency between more than 2 modules also.

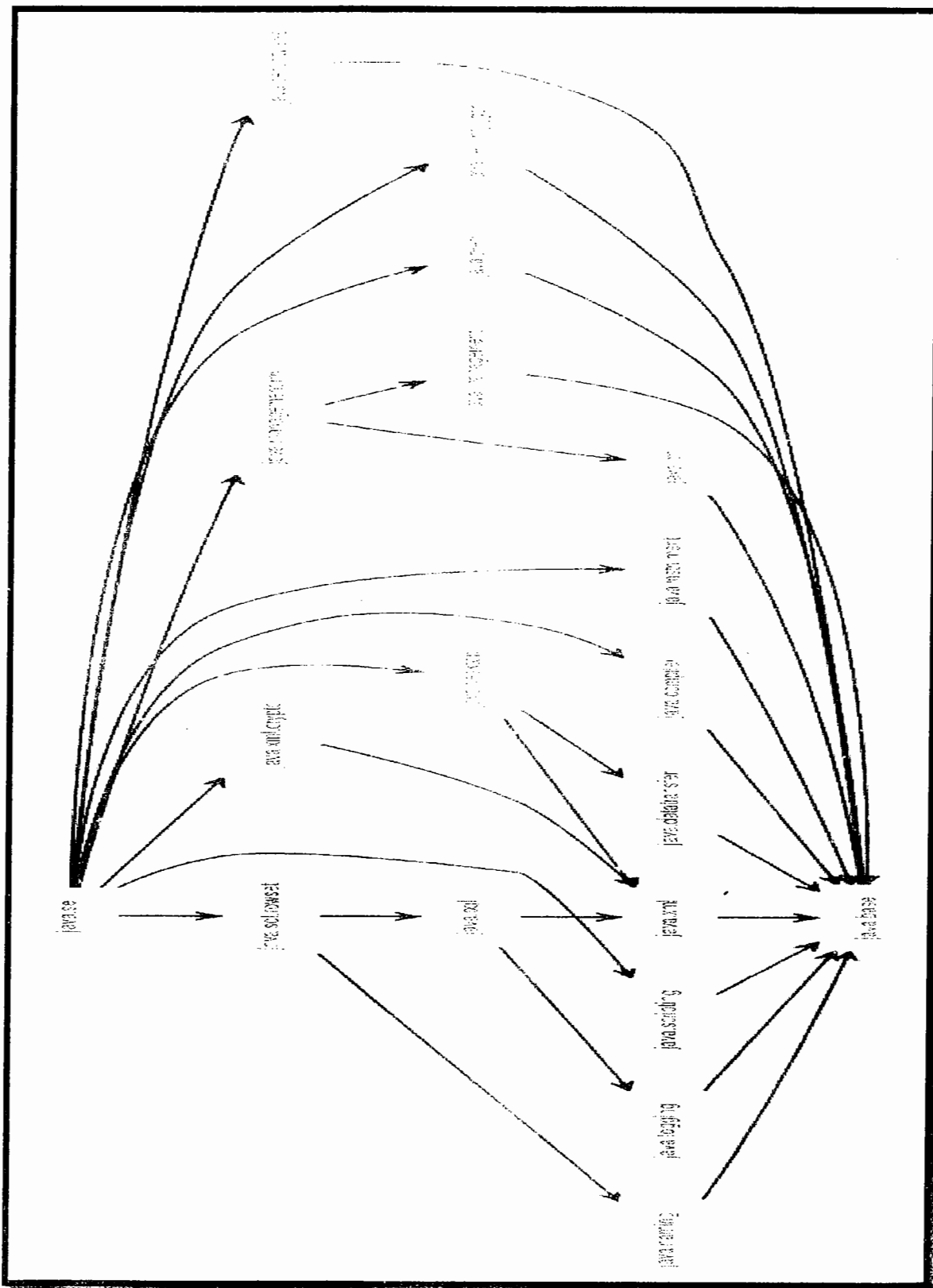
moduleA requires moduleB

moduleB requires moduleC

moduleC requires moduleA



Note: In all predefined modules also, there is no chance of cyclic dependency





Qualified Exports:

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

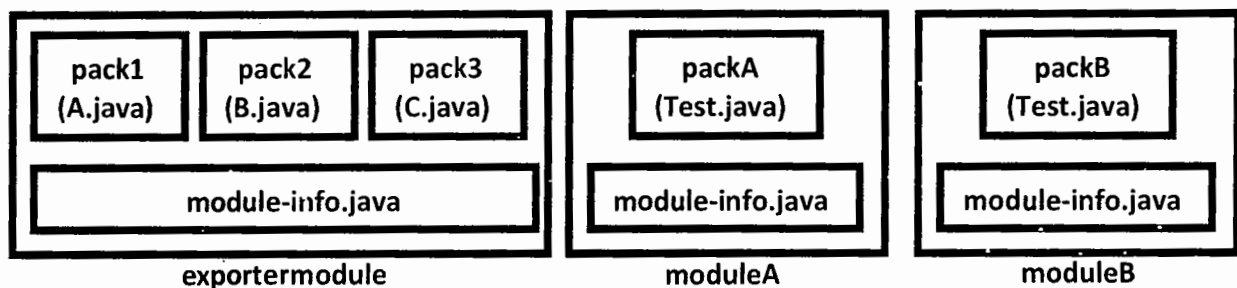
Syntax:

exports <pack1> to <module1>,<module2>,...

Eg:

```
1) module moduleA
2) {
3)   exports pack1;//to export pack1 to all modules
4)   exports pack1 to moduleA;// to export pack1 only for moduleA
5)   exports pack1 to moduleA,moduleB;// to export pack1 for both moduleA,moduleB
6) }
```

Demo Program for Qualified Exports:



Components of exportermodule:

A.java:

```
1) package pack1;
2) public class A
3) {
4) }
```

B.java:

```
1) package pack2;
2) public class B
3) {
4) }
```



C.java:

```
1) package pack3;  
2) public class C  
3) {  
4) }
```

module-info.java:

```
1) module exportermodule  
2) {  
3)     exports pack1;  
4)     exports pack2 to moduleA;  
5)     exports pack3 to moduleA,moduleB;  
6) }
```

	moduleA	moduleB
pack1	✓	✓
pack2	✓	✗
pack3	✓	✓

Components of moduleA:

Test.java:

```
1) package packA;  
2) import pack1.A;  
3) import pack2.B;  
4) import pack3.C;  
5) public class Test  
6) {  
7)     public static void main(String[] args)  
8)     {  
9)         System.out.println("Qualified Exports Demo");  
10)    }  
11)}
```

module-info.java:

```
1) module moduleA  
2) {  
3)     requires exportermodule;  
4) }
```

Explanation:

For moduleA, all 3 packages are available. Hence we can compile and run moduleA successfully.

C:\Users\Durga\Desktop>javac -module-source-path src -d out -m exportermodule, moduleA

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

Qualified Exports Demo



Components of moduleB:

Test.java:

```
1) package packB;
2) import pack1.A;
3) import pack2.B;
4) import pack3.C;
5) public class Test
6) {
7)     public static void main(String[] args)
8)     {
9)         System.out.println("Qualified Exports Demo");
10)    }
11)}
```

module-info.java:

```
1) module moduleB
2) {
3)     requires exportermodule;
4) }
```

Explanation:

For moduleB, only pack1 and pack3 are available. pack2 is not available. But in moduleB we are trying to access pack2 and hence we will get compile time error.

```
C:\Users\Durga\Desktop>javac --module-source-path src -d out -m exportermodule, moduleB
src\moduleB\packB\Test.java:3: error: package pack2 is not visible
import pack2.B;
    ^
```

(package pack2 is declared in module exportermodule, which does not export it to module moduleB)

1 error

Q. Which of the following directives are valid inside module-info.java:

1. requires moduleA;
2. requires moduleA,moduleB;
3. requires moduleA.pack1;
4. requires moduleA.pack1.A;
5. requires static moduleA;
6. requires transitive moduleA;
7. exports pack1;
8. exports pack1,pack2;
9. exports moduleA;



- 10. exports moduleA.pack1.A;
- 11. exports pack1 to moduleA;
- 12. exports pack1 to moduleA,moduleB;

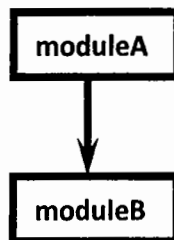
Answers: 1,5,6,7,11,12

Module Graph:

The dependencies between the modules can be represented by using a special graph, which is nothing but Module Graph.

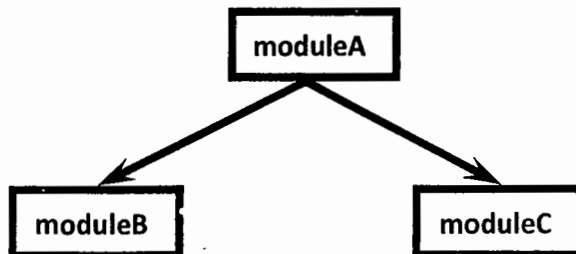
Eg1: If moduleA requires moduleB then the corresponding module graph is :

- 1) module moduleA
- 2) {
- 3) requires moduleB;
- 4) }



Eg 2: If moduleA requires moduleB and moduleC then the corresponding module graph is:

- 1) module moduleA
- 2) {
- 3) requires moduleB;
- 4) requires moduleC;
- 5) }

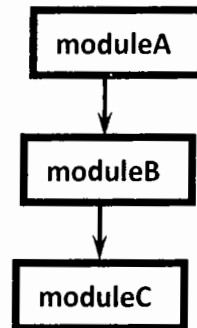


Eg 3: If moduleA requires moduleB and moduleB requires moduleC then the corresponding module graph is:

- 1) module moduleA
- 2) {
- 3) requires moduleB;
- 4) }

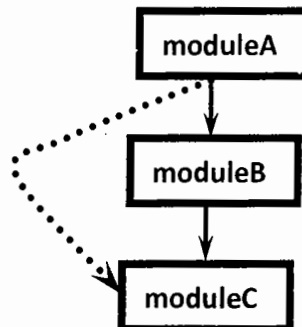


```
5) module moduleB
6) {
7)   requires moduleC;
8) }
```



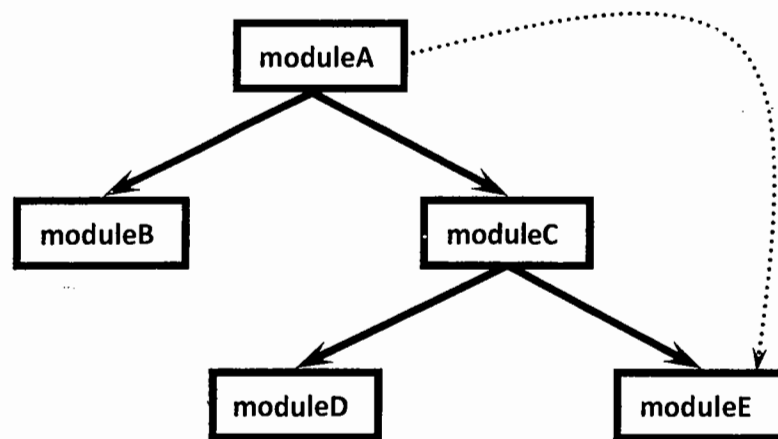
Eg 4: If moduleA requires moduleB and moduleB requires transitive moduleC then the corresponding module graph is:

```
1) module moduleA
2) {
3)   requires moduleB;
4) }
5) module moduleB
6) {
7)   requires transitive moduleC;
8) }
```



Eg 5: If moduleA requires moduleB and moduleC, moduleC requires moduleD and transitive moduleE then the corresponding Modular Graph is:

```
1) module moduleA
2) {
3)   requires moduleB;
4)   requires moduleC;
5) }
6) module moduleC
7) {
8)   requires moduleD;
9)   requires transitive moduleE;
10) }
```

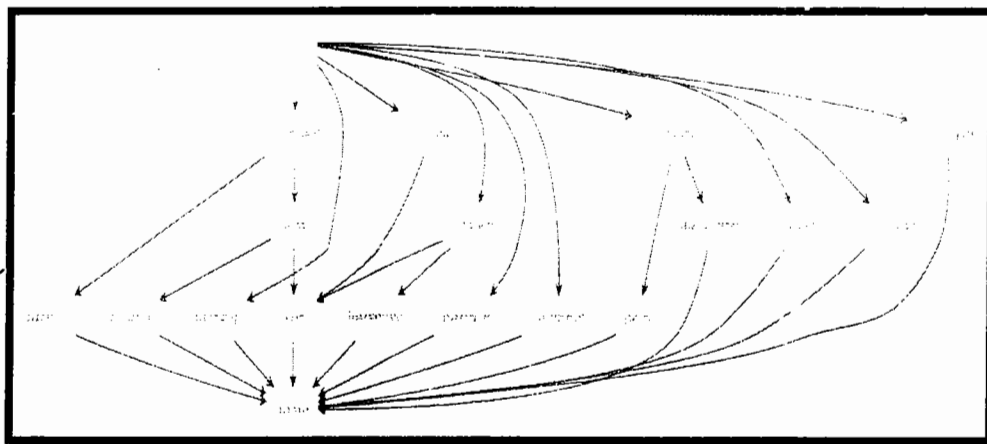


Java 9 JDK itself modularized. All classes of Java SE are divided into several modules.

Eg:

java.base
java.sql
java.xml
java.rmi
etc...

The module graph of JDK is



In the above diagram all modules requires java.base module either directly or indirectly. Hence this module acts as BASE module for all java modules.

Observe modular graphs carefully: java.se and java.sql modules etc

Rules of Module Graph:

1. Two modules with the same name is not allowed.
2. Cyclic Dependency is not allowed between the modules and hence Module Graph should not contain cycles.



Observable Modules:

The modules which are observed by JVM at runtime are called Observable modules.

The modules we are specifying with --module-path option with java command are observed by JVM and hence these are observable modules.

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

The modules present in module-path out are observable modules

JDK itself contains several modules (like java.base, java.sql, java.rmi etc). These modules can be observed automatically by JVM at runtime and we are not required to use --module-path. Hence these are observable modules.

Observable Modules = All Predefined JDK Modules + The modules specified with --module-path option

We can list out all Observable Modules by using --list-modules option with java command.

Eg1: To print all readymade compiled modules (pre defined modules) present in Java 9

```
C:\Users\Durga\Desktop>java --list-modules
java.activation@9
java.base@9
java.compiler@9
....
```

Eg2: Assume our own created compiled modules are available in out folder. To list out these modules including readymade java modules

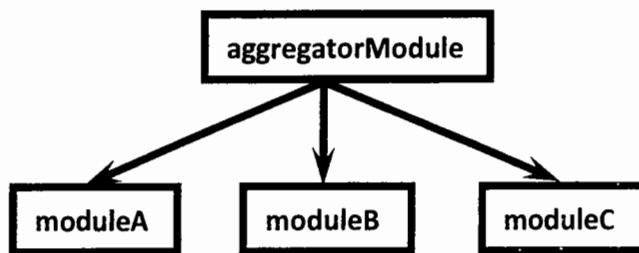
```
C:\Users\Durga\Desktop>java --module-path out --list-modules
java.activation@9
java.base@9
...
exportermodule file:///C:/Users/Durga/Desktop/out/exportermodule/
moduleA file:///C:/Users/Durga/Desktop/out/moduleA/
```



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 Aggregator module. If any module reads aggregator module then automatically all its modules 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.



```
1) module aggregatorModule
2) {
3)   requires transitive moduleA;
4)   requires transitive moduleB;
5)   requires transitive moduleC;
6) }
```

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

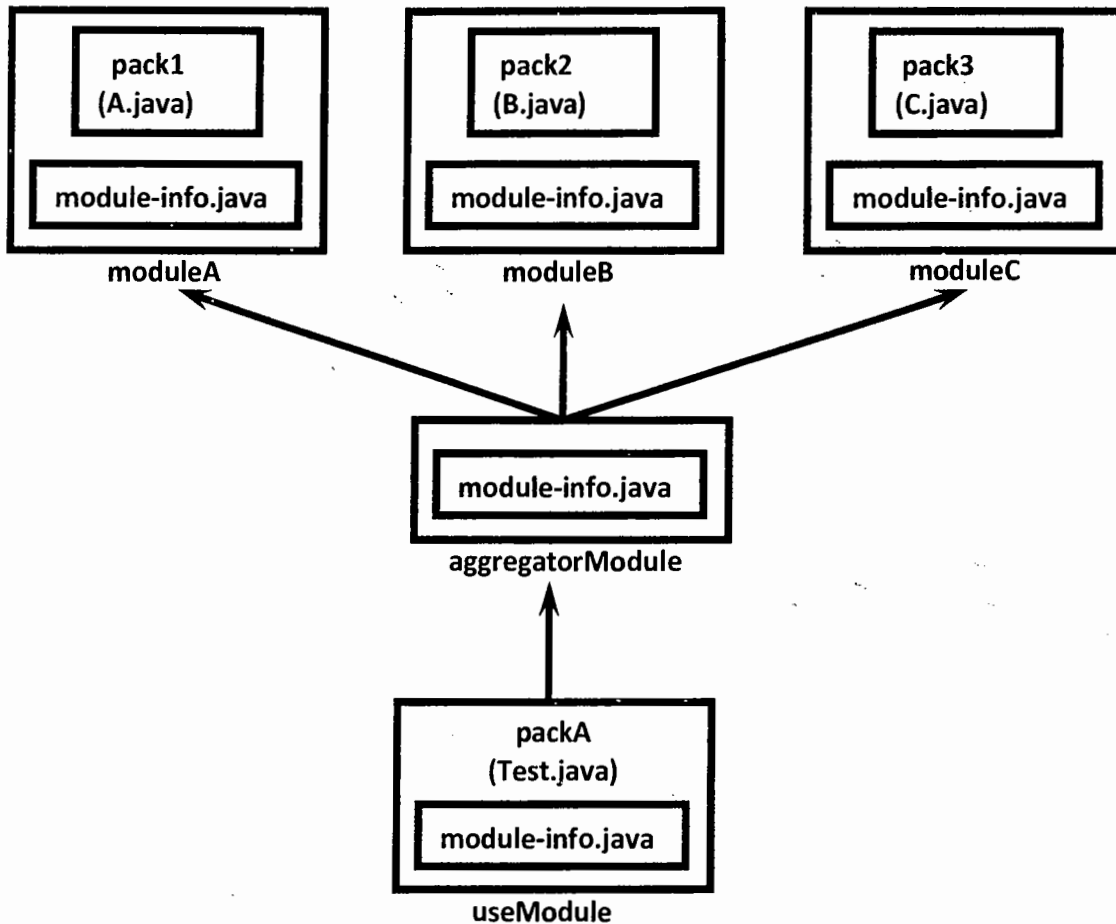
If any module reads aggregatorModule automatically all 3 modules are by default available to that module also.

```
1) module useModule
2) {
3)   requires aggregatorModule;
4) }
```

Now useModule can use functionality of all 3 modules moduleA, moduleB and moduleC.



Demo Program for Aggregator Module:



moduleA components:

A.java:

```
1) package pack1;
2) public class A
3) {
4)     public void m1()
5)     {
6)         System.out.println("moduleA method");
7)     }
8) }
```

module-info.java:

```
1) module moduleA
2) {
3)     exports pack1;
4) }
```



moduleB components:

B.java:

```
1) package pack2;
2) public class B
3) {
4)     public void m1()
5)     {
6)         System.out.println("moduleB method");
7)     }
8) }
```

module-info.java:

```
1) module moduleB
2) {
3)     exports pack2;
4) }
```

moduleC components:

C.java:

```
1) package pack3;
2) public class C
3) {
4)     public void m1()
5)     {
6)         System.out.println("moduleC method");
7)     }
8) }
```

module-info.java:

```
1) module moduleC
2) {
3)     exports pack3;
4) }
```



aggregatorModule components:

module-info.java:

```
1) module aggregatorModule
2) {
3)     requires transitive moduleA;
4)     requires transitive moduleB;
5)     requires transitive moduleC;
6) }
```

useModule components:

Test.java:

```
1) package packA;
2) import pack1.A;
3) import pack2.B;
4) import pack3.C;
5) public class Test
6) {
7)     public static void main(String[] args)
8)     {
9)         System.out.println("Aggregator Module Demo");
10)        A a = new A();
11)        a.m1();
12)
13)        B b = new B();
14)        b.m1();
15)
16)        C c = new C();
17)        c.m1();
18)    }
19) }
```

module-info.java:

Here we are not required to use requires directive for every module, just we have to use requires only for aggregatorModule.

```
1) module useModule
2) {
3)     //requires moduleA;
4)     //requires moduleB;
5)     //requires moduleC;
6)     requires aggregatorModule;
7) }
```



```
C:\Users\Durga\Desktop>javac --module-source-path src -d out -m  
moduleA,moduleB,moduleC,aggregatorModule,useModule
```

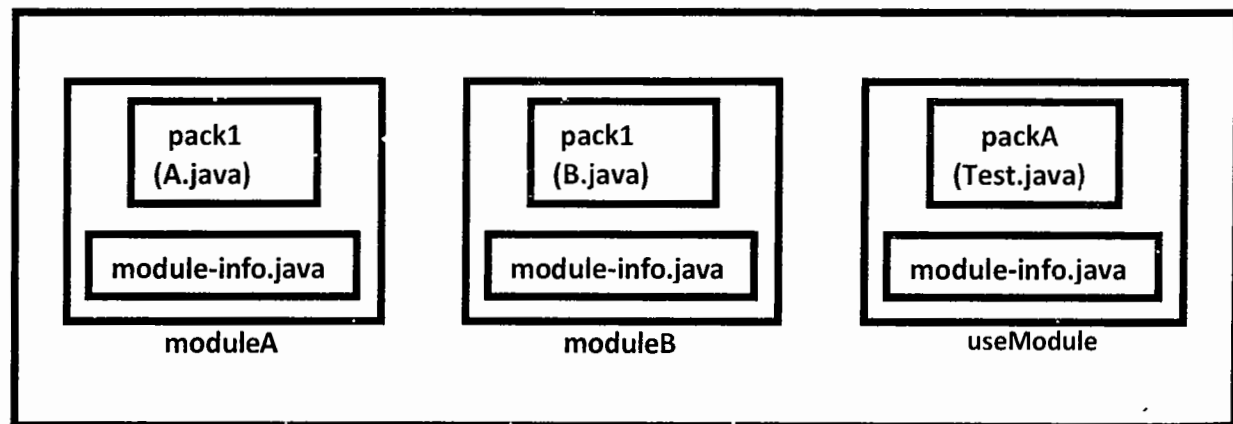
```
C:\Users\Durga\Desktop>java --module-path out -m useModule/packA.Test  
Aggregator Module Demo  
moduleA method  
moduleB method  
moduleC method
```

Package Naming Conflicts:

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

But in Java 9 module System, two modules cannot 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 behavior of the program.

Demo Program:



moduleA components:

A.java:

```
1) package pack1;  
2) public class A  
3) {  
4)     public void m1()  
5)     {  
6)         System.out.println("moduleA method");  
7)     }  
8) }
```



module-info.java:

```
1) module moduleA
2) {
3)     exports pack1;
4) }
```

moduleB components:

B.java:

```
1) package pack1;
2) public class B
3) {
4)     public void m1()
5)     {
6)         System.out.println("moduleB method");
7)     }
8) }
```

module-info.java:

```
1) module moduleB
2) {
3)     exports pack1;
4) }
```

useModule components:

Test.java:

```
1) package packA;
2) public class Test
3) {
4)     public static void main(String[] args)
5)     {
6)         System.out.println("Package Naming Conflicts");
7)     }
8) }
```

module-info.java:

```
1) module useModule {
2)     requires moduleA;
3)     requires moduleB;
4) }
```



```
javac --module-source-path src -d out -m moduleA,moduleB,useModule  
error: module useModule reads package pack1 from both moduleA and moduleB
```

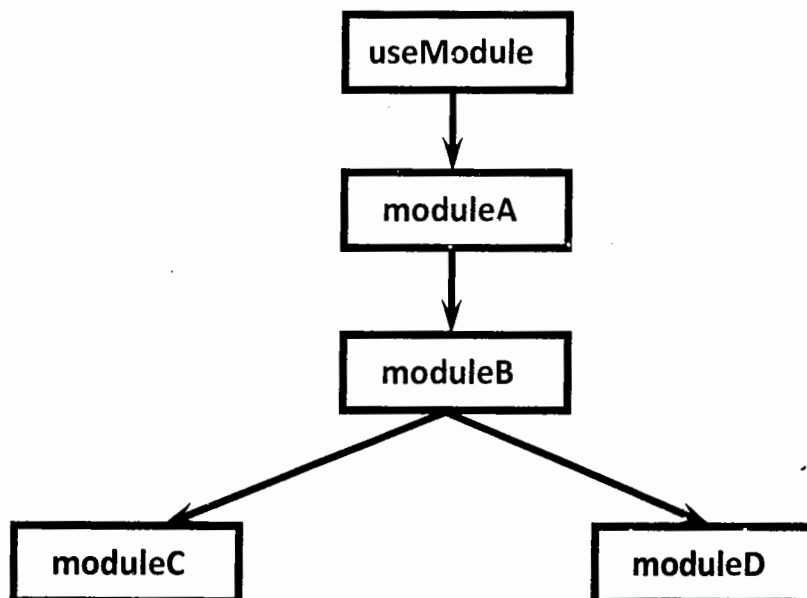
Two modules cannot contain a package with same name.

Module Resolution Process (MRP):

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 at the beginning only JVM will identify and won't start its execution. Hence in modular programming, there is no chance of getting `NoClassDefFoundError` in the middle of program execution.

Demo Program:





Components of useModule:

Test.java:

```
1) package packA;  
2) public class Test  
3) {  
4)     public static void main(String[] args)  
5)     {  
6)         System.out.println("Module Resolution Process(MRP) Demo");  
7)     }  
8) }
```

module-info.java:

```
1) module useModule  
2) {  
3)     requires moduleA;  
4) }
```

Components of moduleA:

module-info.java:

```
1) module moduleA  
2) {  
3)     requires moduleB;  
4) }
```

Components of moduleB:

module-info.java:

```
1) module moduleB  
2) {  
3)     requires moduleC;  
4)     requires moduleD;  
5) }
```

Components of moduleC:

module-info.java:

```
1) module moduleC  
2) {  
3)  
4) }
```



Components of moduleD:

module-info.java:

1. `module moduleD`
2. `{`
- 3.
4. `}`

```
javac --module-source-path src -d out -m moduleA,moduleB,moduleC,moduleD,useModule
```

```
java --module-path out --show-module-resolution -m useModule/packA.Test
```

The module what we are trying to execute will become 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 error if any dependent module is not available.
2. We will get error if multiple modules with the same name
3. We will get error if any cyclic dependency
4. We will get error if two modules contain packages with the same name.

Note:

The following are restricted keywords in java 9:

module, requires, transitive, exports

In normal Java program no restrictions and we can use for identifier purpose also.



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:

```
1) module demoModule
2) {
3) }
```



Test.java:

```
1) package packA;  
2) public class Test  
3) {  
4)     public static void main(String[] args)  
5)     {  
6)         System.out.println("JLINK Demo To create our own customized & small JRE");  
7)     }  
8) }
```

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

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 out folder)

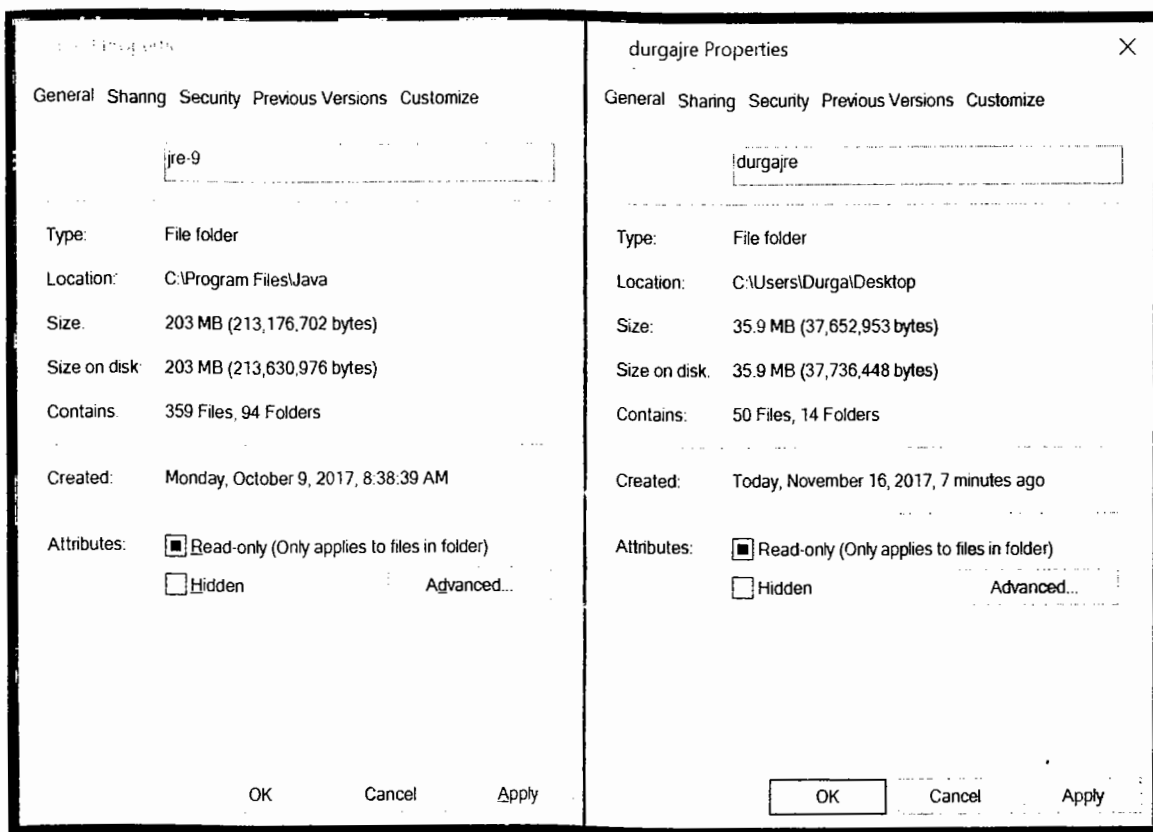
```
out  
|-java.base.jmod  
|-demoModule  
  |-module-info.class  
  |-packA  
  |-Test.class
```

Now we can create our own JRE with JLINK command

```
C:\Users\Durga\Desktop>jlink --module-path out --add-modules demoModule,java.base --output  
durgajre
```



Now observe the size of durgajre is just 35.9MB which is very small when compared with default JRE size 203MB.



We can run our application with our own custom jre (durgajre) as follows

C:\Users\Durga\Desktop\durgajre\bin>java -m demoModule/packA.Test
o/p: JLINK Demo To create our own customized & small JRE

Compressing the size of JRE with compress plugin:

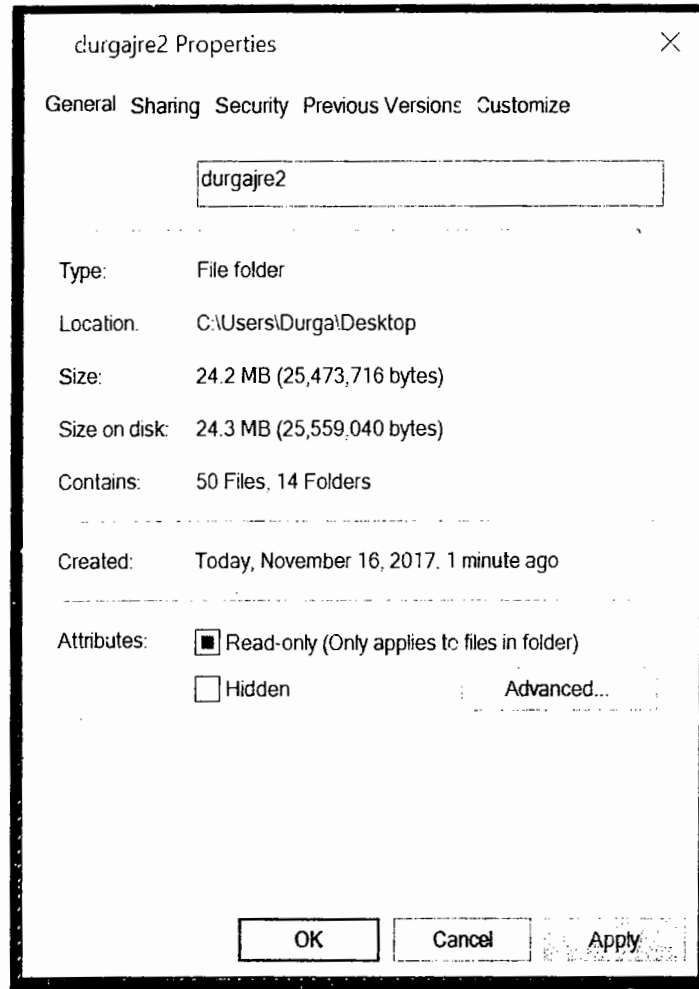
Still we can compress the size of JRE with compress plugin.

```
Command Prompt
C:\Users\Durga\Desktop>jlink --help
Usage: jlink [options] --module-path <modulepath> --add-modules <module>[,<module>...].
Possible options include:
  --add-modules <mod>[,<mod>...]  Root modules to resolve
  --add-services                   Link in service provider modules and
                                   their dependencies
  -c, --compress=<0|1|2>          Enable compression of resources:
                                   level 0: No compression
                                   level 1: Constant string sharing
                                   level 2: ZIP
  --disable-plugin <pluginname>   Disable the plugin mentioned
  --odian <little|big>            Byte order of generated image
```



```
C:\Users\Durga\Desktop>jlink --module-path out --add-modules demoModule,java.base --compress 2 --output durgajre2
```

```
C:\Users\Durga\Desktop\durgajre2\bin>java -m demoModule/packA.Test  
o/p: JLINK Demo To create our own customized & small JRE
```



Providing our own name to the application with launcher plugin:

```
C:\Users\Durga\Desktop>jlink --module-path out --add-modules demoModule,java.base --launcher demoapp=demoModule/packA.Test --compress 2 --output durgajre3
```

Now we can run our application only with the name demoapp

```
C:\Users\Durga\Desktop\durgajre3\bin>demoapp  
JLINK Demo To create our own customized & small JRE
```

If we set the path **PATH = C:\Users\Durga\Desktop\durgajre3\bin**

Then we can run our application from anywhere.

```
D:\>demoapp
```

```
E:\>demoapp
```



Process API Updates (JEP-102)

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.

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

What's New in Java 9 Process API:

1. Added several new methods (like pid(),info() etc) to Process class.
2. Added several new methods (like startPipeline()) to ProcessBuilder class. We can use ProcessBuilder class to create operating system processes.
3. 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
4. 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

1. `ProcessHandle handle=ProcessHandle.current();`
Returns the ProcessHandle of current running Process
2. `ProcessHandle handle=p.toHandle();`
Returns the ProcessHandle of specified Process object.
3. `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

```
1) public class Test
2) {
3)     public static void main(String[] args) throws Exception
4)     {
5)         ProcessHandle p=ProcessHandle.current();
6)         long pid=p.pid();
7)         System.out.println("The PID of current running JVM instance :"+pid);
8)         Thread.sleep(100000);
9)     }
10) }
```

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

ProcessHandle.Info:

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

```
ProcessHandle p = ProcessHandle.current();
ProcessHandle.Info info = p.info();
```

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

1. user():

Return the user of the process.
`public Optional<String> user()`

2. command():

Returns the command, that can be used to start the process.
`public Optional<String> command()`



3. startInstant():

Returns the start time of the process.

public Optional<String> startInstant()

4. totalCpuDuration():

Returns the total cputime accumulated of the process.

public Optional<String> totalCpuDuration()

Use Case-2: To get snapshot of the current running process info

```
1) public class Test
2) {
3)     public static void main(String[] args) throws Exception
4)     {
5)         ProcessHandle p=ProcessHandle.current();
6)         ProcessHandle.Info info=p.info();
7)         System.out.println("Complete Process Inforamtion:\n"+info);
8)         System.out.println("User: "+info.user().get());
9)         System.out.println("Command: "+info.command().get());
10)        System.out.println("Start Time: "+info.startInstant().get());
11)        System.out.println("Total CPU Time Acquired: "+info.totalCpuDuration().get());
12)    }
13) }
```

Use Case-3: To get snapshot of the Particular Process Based on id

```
1) import java.util.*;
2) public class Test
3) {
4)     public static void main(String[] args) throws Exception
5)     {
6)         Optional<ProcessHandle> opt=ProcessHandle.of(7532);
7)         ProcessHandle p=opt.get();
8)         ProcessHandle.Info info=p.info();
9)         System.out.println("Complete Process Inforamtion:\n"+info);
10)        System.out.println("User: "+info.user().get());
11)        System.out.println("Command: "+info.command().get());
12)        System.out.println("Start Time: "+info.startInstant().get());
13)        System.out.println("Total CPU Time Acquired: "+info.totalCpuDuration().get());
14)    }
15) }
```



ProcessBuilder:

We can use ProcessBuilder to create processes.

We can create ProcessBuilder object by using the following constructor.

```
public ProcessBuilder(String... command)
```

The argument should be valid command to invoke the process.

Eg:

```
ProcessBuilder pb = new ProcessBuilder("javac", "Test.java");  
ProcessBuilder pb = new ProcessBuilder("java", "Test");  
ProcessBuilder pb = new ProcessBuilder("notepad.exe", "D:\\names.txt");
```

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

```
pb.start();
```

Use Case-4: To create and Start a process by using ProcessBuilder

FrameDemo.java:

```
1) import java.awt.*;  
2) import java.awt.event.*;  
3) public class FrameDemo  
4) {  
5)     public static void main(String[] args)  
6)     {  
7)         Frame f = new Frame();  
8)         f.addWindowListener( new WindowAdapter()  
9)         {  
10)             public void windowClosing(WindowEvent e)  
11)             {  
12)                 System.exit(0);  
13)             }  
14)         });  
15)         f.add(new Label("This Process Started from Java by using ProcessBuilder !!!"));  
16)         f.setSize(500,500);  
17)         f.setVisible(true);  
18)     }  
19) }
```




Test.java

```
1) public class Test
2) {
3)     public static void main(String[] args) throws Exception
4)     {
5)         ProcessBuilder pb=new ProcessBuilder("java","FrameDemo");
6)         pb.start();
7)     }
8) }
```

Use Case-5: To open a file with notepad from java by using ProcessBuilder

```
1) public class Test
2) {
3)     public static void main(String[] args) throws Exception
4)     {
5)         new ProcessBuilder("notepad.exe","FrameDemo.java").start();
6)     }
7) }
```

Use Case-6: To start and destroy a process from java by using ProcessBuilder

```
1) class Test
2) {
3)     public static void main(String[] args) throws Exception
4)     {
5)         ProcessBuilder pb=new ProcessBuilder("java","FrameDemo");
6)         Process p=pb.start();
7)         System.out.println("Process Started with id:"+p.pid());
8)         Thread.sleep(10000);
9)         System.out.println("Destroying the process with id:"+p.pid());
10)        p.destroy();
11)    }
12) }
```

Use Case-7: To destroy a process which is not created from Java

```
1) import java.util.*;
2) class Test
3) {
4)     public static void main(String[] args) throws Exception {
5)         Optional<ProcessHandle> optph=ProcessHandle.of(5232);
6)         ProcessHandle ph=optph.get();
7)         ph.destroy();
8)     }
9) }
```



Use Case-8: To display all running process information

```
1) import java.util.*;
2) import java.util.stream.*;
3) import java.time.*;
4) class Test
5) {
6)     public static void dumpProcessInfo(ProcessHandle p)
7)     {
8)         ProcessHandle.Info info=p.info();
9)         System.out.println("Process Id:"+p.pid());
10)        System.out.println("User: "+info.user().orElse(""));
11)        System.out.println("Command: "+info.command().orElse(""));
12)        System.out.println("Start Time: "+info.startInstant().orElse(Instant.now()).toString());
13)        System.out.println("Total CPU Time Acquired: "+info.totalCpuDuration().
14)                               orElse(Duration.ofMillis(0)).toMillis());
15)        System.out.println();
16)    }
17)    public static void main(String[] args) throws Exception
18)    {
19)        Stream<ProcessHandle> allp=ProcessHandle.allProcesses();
20)        allp.limit(100).forEach(ph->dumpProcessInfo(ph));
21)    }
22) }
```

Use Case-9: To display all child process information

```
1) import java.util.stream.*;
2) import java.time.*;
3) class Test
4) {
5)     public static void dumpProcessInfo(ProcessHandle p) {
6)         ProcessHandle.Info info=p.info();
7)         System.out.println("Process Id:"+p.pid());
8)         System.out.println("User: "+info.user().orElse(""));
9)         System.out.println("Command: "+info.command().orElse(""));
10)        System.out.println("Start Time: "+info.startInstant().orElse(Instant.now()).toString());
11)        System.out.println("Total CPU Time Acquired: "+info.totalCpuDuration()
12)                               .orElse(Duration.ofMillis(0)).toMillis());
13)        System.out.println();
14)    }
15)    public static void main(String[] args) throws Exception {
16)        ProcessHandle handle=ProcessHandle.current();
17)        Stream<ProcessHandle> childp=handle.children();
18)        childp.forEach(ph->dumpProcessInfo(ph));
19)    }
20) }
```



Note: If Current Process not having any child processes then we won't get any output

Use Case-10: To perform a task at the time of Process Termination

```
1) import java.util.concurrent.*;
2) public class Test
3) {
4)     public static void main(String[] args) throws Exception
5)     {
6)         ProcessBuilder pb=new ProcessBuilder("java","FrameDemo");
7)         Process p=pb.start();
8)         System.out.println("Process Started with id:"+p.pid());
9)         CompletableFuture<Process> future=p.onExit();
10)        future.thenAccept(p1->System.out.println("Process Terminated with Id:"+p1.pid()));
11)        System.out.println(future.get());
12)    }
13) }
```

Output [In Normal Termination]:

```
D:\durga_classes>java Test
Process Started with id:4828
Process[pid=4828, exitValue=0]
Process Terminated with Id:4828
```

Output [In Abnormal Termination alt+ctrl+delete]:

```
D:\durga_classes>java Test
Process Started with id:12512
Process[pid=12512, exitValue=1]
Process Terminated with Id:12512
```

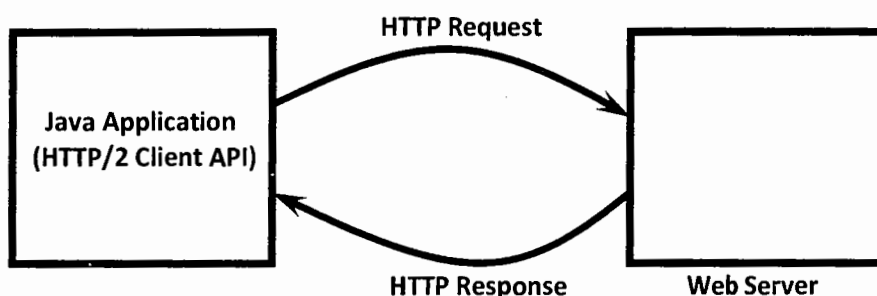
Observe exit value: 0 for normal termination and non-zero for abnormal termination



HTTP/2 Client

What is the purpose of 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.

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.

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



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...

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:

1. `HttpClient`
2. `HttpRequest`
3. `HttpResponse`

Note:

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

```
1) module demoModule
2) {
3)   requires jdk.incubator.httpclient;
4) }
```

Steps to send Http Request and process Http Response from Java Application:

1. Create `HttpClient` Object
2. Create `HttpRequest` object
3. Send `HttpRequest` by using `HttpClient` and Get the `HttpResponse`
4. 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:

1. send() to send synchronous request(blocking mode)
2. sendAsync() to send Asynchronous Request(Non Blocking Mode)

Eg:

```
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.

Status Line
Response Headers
Response Body

Structure of HTTP Response

HttpResponse class contains the following methods retrieve data from the response

1. statusCode()

Returns status code of the response

It may be (1XX,2XX,3XX,4XX,5XX)

2. body()

Returns body of the response



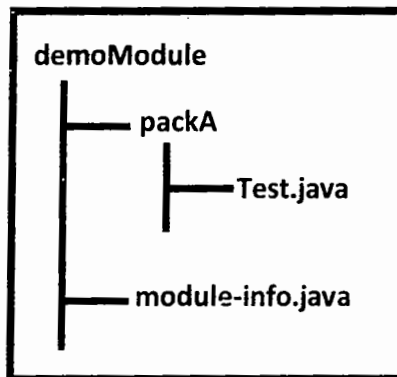
3. headers()

Returns header information of the response

Eg:

```
System.out.println("Status Code:"+resp.statusCode());
System.out.println("Body:"+resp.body());
System.out.println("Response Headers Info");
HttpHeaders header=resp.headers();
Map<String,List<String>> map=header.map();
map.forEach((k,v)->System.out.println("\t"+k+"-"+v));
```

Demo Program to send GET Request in Blocking Mode:



module-info.java:

```
1) module demoModule
2) {
3)     requires jdk.incubator.httpclient;
4) }
```

Test.java:

```
1) package packA;
2) import jdk.incubator.http.HttpClient;
3) import jdk.incubator.http.HttpRequest;
4) import jdk.incubator.http.HttpResponse;
5) import jdk.incubator.http.HttpHeaders;
6) import java.net.URI;
7) import java.util.Map;
8) import java.util.List;
9) public class Test
10) {
11)     public static void main(String[] args) throws Exception
12)     {
13)         String url="https://www.redbus.in/info/aboutus";
14)         sendGetSyncRequest(url);
```



```
15) }
16) public static void sendGetSyncRequest(String url) throws Exception
17) {
18)     HttpClient client=HttpClient.newHttpClient();
19)     HttpRequest req=HttpRequest.newBuilder(new URI(url)).GET().build();
20)     HttpResponse resp=client.send(req,HttpResponse.BodyHandler.asString());
21)     processResponse(resp);
22) }
23) public static void processResponse(HttpResponse resp)
24) {
25)     System.out.println("Status Code:"+resp.statusCode());
26)     System.out.println("Response Body:"+resp.body());
27)     HttpHeaders header=resp.headers();
28)     Map<String,List<String>> map=header.map();
29)     System.out.println("Response Headers");
30)     map.forEach((k,v)->System.out.println("\t"+k+": "+v));
31) }
32) }
```

Note:

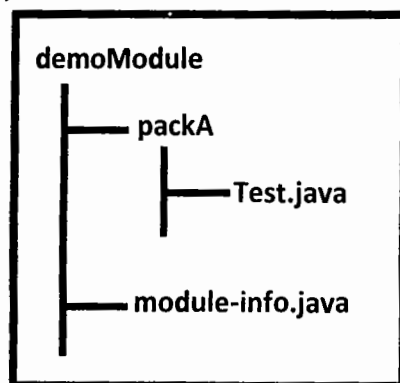
Writing Http Response body to file abc.html:

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

Paths is a class present in java.nio.file package and hence we should write import as
import java.nio.file.Paths;

In this case, abc.html file will be created in the current working directory which contains total response body.

Demo Program:





module-info.java:

```
1) module demoModule
2) {
3)     requires jdk.incubator.httpclient;
4) }
```

Test.java:

```
1) package packA;
2) import jdk.incubator.http.HttpClient;
3) import jdk.incubator.http.HttpRequest;
4) import jdk.incubator.http.HttpResponse;
5) import jdk.incubator.http.HttpHeaders;
6) import java.net.URI;
7) import java.util.Map;
8) import java.util.List;
9) import java.nio.file.Paths;
10) public class Test
11) {
12)     public static void main(String[] args) throws Exception
13)     {
14)         String url="https://www.redbus.in/info/aboutus";
15)         sendGetSyncRequest(url);
16)     }
17)     public static void sendGetSyncRequest(String url) throws Exception
18)     {
19)         HttpClient client=HttpClient.newHttpClient();
20)         HttpRequest req=HttpRequest.newBuilder(new URI(url)).GET().build();
21)         HttpResponse resp=client.send(req,HttpResponse.BodyHandler.asFile(Paths.get("abc.html")));
22)         processResponse(resp);
23)     }
24)     public static void processResponse(HttpResponse resp)
25)     {
26)         System.out.println("Status Code:"+resp.statusCode());
27)         //System.out.println("Response Body:"+resp.body());
28)         HttpHeaders header=resp.headers();
29)         Map<String,List<String>> map=header.map();
30)         System.out.println("Response Headers");
31)         map.forEach((k,v)->System.out.println("\t"+k+": "+v));
32)     }
33) }
```

abc.html will be created in the current working directory. Open that file to see body of response.



Asynchronous Communication:

In Blocking Mode (Synchronous Mode), Once we send Http Request, we should wait until getting response. It creates performance problems.

But in Non-Blocking Mode (Asynchronous Mode), we are not required to wait until getting the response. We can continue our execution and later point of time we can use that HttpResponse once it is ready, so that performance of the system will be improved.

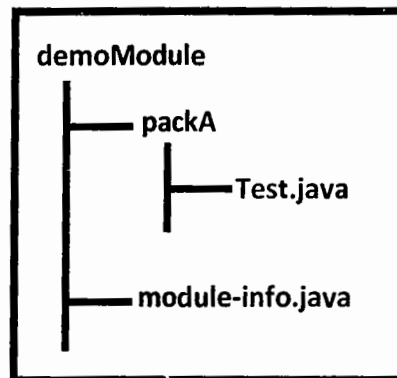
HttpClient class contains `sendAsync()` method to send asynchronous request.

```
CompletableFuture<HttpResponse<String>> cf =  
client.sendAsync(req, HttpResponse.BodyHandler.asString());
```

CompletableFuture Object can be used to hold HttpResponse in asynchronous communication. This class present in `java.util.concurrent` package. This class contains `isDone()` method to check whether processing completed or not.

```
public boolean isDone()
```

Demo Program For Asynchronous Communication:



module-info.java:

```
1) module demoModule  
2) {  
3)   requires jdk.incubator.httpclient;  
4) }
```

Test.java:

```
1) package packA;  
2) import jdk.incubator.http.HttpClient;  
3) import jdk.incubator.http.HttpRequest;  
4) import jdk.incubator.http.HttpResponse;
```



```
5) import jdk.incubator.http.HttpHeaders;
6) import java.net.URI;
7) import java.util.Map;
8) import java.util.List;
9) import java.util.concurrent.CompletableFuture;
10) public class Test
11) {
12)     public static void main(String[] args) throws Exception
13)     {
14)         String url="https://www.redbus.in/info/aboutus";
15)         sendGetAsyncRequest(url);
16)     }
17)     public static void sendGetAsyncRequest(String url) throws Exception
18)     {
19)         HttpClient client=HttpClient.newHttpClient();
20)         HttpRequest req=HttpRequest.newBuilder(new URI(url)).GET().build();
21)         System.out.println("Sending Asynchronous Request...");
22)         CompletableFuture<HttpResponse<String>> cf = client.sendAsync(req,HttpResponse.BodyHandlers.asString());
23)         int count=0;
24)         while(!cf.isDone())
25)         {
26)             System.out.println("Processing not done and doing other activity:" + ++count);
27)         }
28)         processResponse(cf.get());
29)     }
30)     public static void processResponse(HttpResponse resp)
31)     {
32)         System.out.println("Status Code:"+resp.statusCode());
33)         //System.out.println("Response Body:"+resp.body());
34)         HttpHeaders header=resp.headers();
35)         Map<String,List<String>> map=header.map();
36)         System.out.println("Response Headers");
37)         map.forEach((k,v)->System.out.println("\t"+k+" "+v));
38)     }
39) }
```

