JAVA 9 Version Features:
—----------------------

1. Private Methods in Interfaces
2. try-with-resources Enhancements
3. Diamond Operator Enhancement in JAVA.
4. Factory Methods for Unmodifiable Collections.
5. JSHELL
6. JPMS
7. Enhancements in Stream API
8. Process API Updations
9. JLinker

—----

—----

Private Methods in Interfaces
—----------------------------

Before the JAVA 8 version , interfaces were able to allow only abstract methods, from JAVA 8 version onwards interfaces are able to allow static methods, default methods along with abstract methods.

Default method is a method with “default” access modifier inside the interfaces, it will provide initial implementation for any method that implementation classes can override or reuse.

```
public interface I{
      public default void meth(){
            —----
      }
}
```

In Java applications, if we want to add a new functionality to an interface without affecting implementation classes then we have to use “Default Methods”.

EX:

```
public interface Calculator{
      public int add(int i, int j);
      public default int sub(int i, int j){
            ------
      }
}
public class CalculatorImpl implements Calculator{
      public int add(int i, int j){
            ------
      }
}
```

In the JAVA 9 version we are able to declare private methods inside the interfaces along with static methods , default methods and abstract methods.

The main intention of the private methods inside the interfaces is to improve code reusability in the default methods and to utilize up to interface only, not to utilize in the implementation classes.

EX:

```
interface Transaction{
   private void preTransaction(){
        System.out.println("Open Account Database.....");
        System.out.println("Begin Transaction.....");
   }
   private  void postTransaction(){
        System.out.println("Commit / Rollback
Transation.....");
        System.out.println("End Transaction......");
        System.out.println("Close Account
Database......");
   }
   public default void deposit(){
        preTransaction();
        System.out.println("*****Deposit Logic******");
        postTransaction();
   }
   public default void withdraw(){
      preTransaction();
```

```
        System.out.println("*****Withdraw Logic******");
      postTransaction();
    }
    public default void transferFunds(){
        preTransaction();
        System.out.println("*****Transafer Funds
Logic******");
       postTransaction();
    }
}
class TransactionImpl implements Transaction{

}
public class Main {
    public static void main(String[] args) {
        Transaction transaction = new TransactionImpl();
        transaction.deposit();
        System.out.println();

        transaction.withdraw();
        System.out.println();

        transaction.transferFunds();
    }
}
```

```
Open Account Database.....
Begin Transaction.....
*****Deposit Logic******
Commit / Rollback Transation.....
End Transaction......
Close Account Database......

Open Account Database.....
Begin Transaction.....
*****Withdraw Logic******
Commit / Rollback Transation.....
End Transaction......
Close Account Database......
```

```
Open Account Database.....
Begin Transaction.....
*****Transafer Funds Logic******
Commit / Rollback Transation.....
End Transaction......
Close Account Database......
```

try-with-resources Enhancements:

In general, in Java applications, we may use resources like Streams, Sockets, Database Connections,.... When we perform operations with these resources we may get exceptions, to handle these exceptions if we use try-catch-finally then we have to use the following conventions.

1. Declare the resources before try block.
2. Create the resources inside the try block.
3. Close the resources inside the finally block.

```
public class Test{
      public static void main(String[] args){
            BufferedReader br = null;
            Socket s = null;
            Connection con = null;
            try{
                  br = new BufferedReader(new InputStreamReader(System.in));
                  S = new Socket("localhost",4444);
                  con = DriverManager.getConnection(----);
                  ------
            }catch(Exception e){
                  e.printStackTrace();
            }finally{
                  try{
                        br.close();
                        s.close();
                        con.close();
                  }catch(Exception e){
                        e.printStackTrace();
                  }
            }
      }
}
```

The above conventions are providing the following two problems

1. Developers must close the resources explicitly, it is not guaranteed.
2. It will increase confusion when we write try-catch-finally inside the finally block in order to write close() methods inside the finally block.

To overcome these problems, JAVA 7 version has provided a new enhancement in try-catch-finally syntax that is try-with-resources or Auto-closeable resources.

In try-with-resources, JVM will close all the resources automatically when JVM is coming out from try block.

Syntax:

```
try(Resource-1; Resource-2;....Resource-n;){
      ------
}catch(Exception e){
      ------
}
```

If we want to use Resources along with try-with-resources syntax then that resources must implement java.lang.AutoCloseable marker interface either directly or indirectly.

If we provide the resources along with try in try-with-resources syntax then the resources reference variables are converted to final variables internally.

```
public class Test{
      public static void main(String[] args){
            try(
                  BufferedReader br = new BufferedReader(new
                                    InputStreamReader(System.in));
                  Socket s = new Socket("localhost",4444);
                  Connection con = DriverManager.getConnection(----);

                  ){
                        ------
            }catch(Exception e){
                  e.printStackTrace();
            }
      }
}
```

Up to JAVA 8 version , it is not possible to declare and create resources before try block, we must declare and create resources along with try keyword.

IN JAVA 9 version, there is an enhancement in try-with-resources like to declare and create the resources before try-with-resources syntax and we can pass the resources reference variables as parameters to the try.

Syntax:

```
Resource1 r1 = new Resource1();
Resource1 r2 = new Resource1();
—----
—----
Resource1 rn = new Resource1();

try(r1;r2;....rn;){
}catch(Exception e){
      e.printStackTrace();
}
```

EX:

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
Connection con = DriverManager.getConnection(--);
Socket s = new Socket(---);
try(
br;con;s;
){
—-----
}catch(Exception e){
   e.printStackTrace();
}
```

EX:

```
import java.io.*;

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

       BufferedReader bufferedReader = new BufferedReader(new
InputStreamReader(System.in));
       FileOutputStream fileOutputStream = new
FileOutputStream("E:/abc/xyz/welcome.txt");
```

```
        FileInputStream fileInputStream = new
FileInputStream("E:/abc/xyz/welcome.txt");

        try (bufferedReader;fileOutputStream;fileInputStream) {
            System.out.print("Enter Data  : ");
            String data = bufferedReader.readLine();
            fileOutputStream.write(data.getBytes());
            byte[] btArray = new
byte[fileInputStream.available()];
            fileInputStream.read(btArray);
            System.out.println(new String(btArray));
        }catch (Exception e){
            e.printStackTrace();
        }
    }
}
```

```
Enter Data  : Welcome To Durgasoft
Welcome To Durgasoft
```

Diamond Operator in JAVA:

IN Java applications, arrays are able to allow only homogeneous elements of fixed size in nature, it is not flexible for the developers.

In Java applications, if we want to represent different types of elements or heterogeneous elements in a dynamically growable manner we have to use Collections.

In Collections, we are able to represent heterogeneous elements , it will reduce typed ness in java applications and it is able to provide type unsafe operations.

In Collections, to improve typed ness and to perform type safe operations we have to use generics.

In the case of Generics, we will provide Type Parameters along with Collection classes in order to fix the type of elements which we want to add to the collection objects.

```
CollectionName<T> refVar = new CollectionName<T>();
EX:
ArrayList<String> al1 = new ArrayList<String>();
ArrayList<Integer> al2 = new ArrayList<Integer>();
```

In the above Generic Type Parameter declaration, we must use the same type at both left side Type Parameter and Right side Type Parameter, in this case JAVA7 version has provided a flexibility to remove Type Parameter at right side of the expression in <>, here empty <> is called diamond operator.

```
CollectionName<T> refVar = new CollectionName<>();
EX:
ArrayList<String> al1 = new ArrayList<>();
ArrayList<Integer> al2 = new ArrayList<>();
```

Up to JAVA 8 version, we are able to use <> for the Collection classes in the generic classes declaration and in the Collection classes objects creation, but it is not possible to use <> operator to the anonymous inner classes.

In the above context, JAVA 9 version has provided an enhancement on diamond operator like to apply <> operator for Anonymous inner classes also.

```
Comparator<String> comp = new Comparator<>(){
      -----
};
```

EX:

```java
import java.util.Comparator;
import java.util.TreeSet;

public class Main {
   public static void main(String[] args) throws Exception {
       Comparator<String> comparator = new Comparator<>() {
           @Override
           public int compare(String str1, String str2) {
               return -str1.compareTo(str2);
           }
       };
       TreeSet<String> treeSet = new TreeSet<>(comparator);
       treeSet.add("FFF");
       treeSet.add("AAA");
       treeSet.add("EEE");
```

```
        treeSet.add("BBB");
        treeSet.add("DDD");
        treeSet.add("CCC");
        System.out.println(treeSet);

    }
}
```

Factory Methods for Unmodifiable Collections:
—--
Before JAVA 1.2 version we are able to create Collection object and we are able to perform modifications over the Collection elements as per the requirement, here there is no mechanism to fix the number of elements in the Collection objects and there is no mechanism to protect the collection elements, here protecting COllection elements in the sense not to allow modifications like updations , remove,... operation over the Collection elements.

To overcome the above problems JAVA 1.2 version has provided the following methods in Collections class to make immutable Collection or Unmodifiable Collections.

```
public static List unmodifiableList(List l)
public static Set unmodifiableSet(Set l)
public static Map unmodifiableMap(Map m)
```

After creating Immutable Collections, if we perform modifications over the elements then JVM will raise an exception like
java.lang.UnsupportedOperationException

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

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

        List<String> list = new ArrayList<>();
        list.add("AAA");
        list.add("BBB");
        list.add("CCC");
```

```
        list.add("DDD");
        System.out.println(list);

        list = Collections.unmodifiableList(list);
        System.out.println(list);
        //list.add("EEE"); --->
java.lang.UnsupportedOperationException
        System.out.println(list);

    }
}
```

```
[AAA, BBB, CCC, DDD]
[AAA, BBB, CCC, DDD]
Exception in thread "main" java.lang.UnsupportedOperationException
      at
java.base/java.util.Collections$UnmodifiableCollection.add(Collections.java:1
056)
      at Main.main(Main.java:17)
```

EX:

```
import java.util.Collections;
import java.util.HashSet;
import java.util.Set;

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

        Set<String> set = new HashSet<>();
        set.add("AAA");
        set.add("BBB");
        set.add("CCC");
        set.add("DDD");
        System.out.println(set);
        set = Collections.unmodifiableSet(set);
        System.out.println(set);
        //set.add("EEE"); --->
java.lang.UnsupportedOperationException

    }
}
```

EX:

```
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

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

       Map<Integer, String> map = new HashMap<>();
       map.put(1, "AAA");
       map.put(2, "BBB");
       map.put(3, "CCC");
       map.put(4, "DDD");
       System.out.println(map);

       map = Collections.unmodifiableMap(map);
       System.out.println(map);
       //map.put(3, "XXX"); --->
java.lang.UnsupportedOperationException


   }
}
```

In the JAVA 9 version, Java has provided the following factory methods to make immutable List, Immutable Set and Immutable Map.

```
public static List<T> of(T … t)
public static Set<T> of(T … t)
public static Map<K, V> of(k1, v1, K2, V2,..... K_n, V_n)
```

EX:

```
import java.util.List;
import java.util.Map;
import java.util.Set;

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

       List<Integer> list = List.of(10,20,30,40);
```

```
        System.out.println(list);

        Set<Integer> set = Set.of(20,30,40,50);
        System.out.println(set);

        Map<Integer, String> map =
Map.of(1,"AAA",2,"BBB",3,"CCC",4,"DDD");
        System.out.println(map);

    }
}
```

```
import java.util.List;
import java.util.Map;
import java.util.Set;

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

        List<Integer> list = List.of(10,20,30,40);
        System.out.println(list);
        //list.add(50);-->
java.lang.UnsupportedOperationException

        Set<Integer> set = Set.of(20,30,40,50);
        System.out.println(set);
        //set.add(60);--> java.lang.UnsupportedOperationException

        Map<Integer, String> map =
Map.of(1,"AAA",2,"BBB",3,"CCC",4,"DDD");
        System.out.println(map);
        //map.put(5,"EEE"); --->
java.lang.UnsupportedOperationException

    }
}
```

JSHELL
—-------
JSHELL is an interactive Mode to check code blocks instantly.
JSHELL is also called REPL tool

R : Read
E : Evaluate
P : Program
L : Loop

Note: JSHELL is mainly for checking the code instantly, it is not for the development of the applications.

Before JAVA 9 version, if we want to display a simple message on command prompt then we have to provide main class , main() and System.out.println() statements, but in JSHELL we are able to display the message directly by using System.out.println() .

To Open JSHELL we have to use the following command in the command prompt.

JSHELL
EX:
C:\Users\Administrator>JSHELL
| Welcome to JShell -- Version 17.0.6
| For an introduction type: /help intro

jshell>

To get Introduction to JSHELL we have to use the following command on command prompt.

/help intro

EX:
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 the jshell tool commands that allow you to understand and
|  control what you are doing, like:  /list
|
|  For a list of commands: /help

jshell>
```

If we want to get all the commands which are supported by JSHELL we have to use the following command.

```
/help
```

EX:

```
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
|  /drop <name or id>
|       delete a source entry
|  /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 type declarations
|  /imports
|       list the imported items
|  /exit [<integer-expression-snippet>]
|       exit the jshell tool
```

```
|  /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 the jshell tool
|  /reload [-restore] [-quiet] [-class-path <path>] [-module-path <path>]...
|       reset and replay relevant history -- current or previous (-restore)
|  /history [-all]
|       history of what you have typed
|  /help [<command>|<subject>]
|       get information about using the jshell tool
|  /set editor|start|feedback|mode|prompt|truncation|format ...
|       set configuration information
|  /? [<command>|<subject>]
|       get information about using the jshell tool
|  /!
|       rerun last snippet -- see /help rerun
|  /<id>
|       rerun snippets by ID or ID range -- see /help rerun
|  /-<n>
|       rerun n-th previous snippet -- see /help rerun
|
|  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
|  keys
|       a description of readline-like input editing
|  id
|       a description of snippet IDs and how use them
|  shortcuts
|       a description of keystrokes for snippet and command completion,
|       information access, and automatic code generation
|  context
|       a description of the evaluation context options for /env /reload and
/reset
|  rerun
|       a description of ways to re-evaluate previously entered snippets
```

```
jshell>

IN JSHELL, we are able to display messages directly, we are able to evaluate
the expressions directly.
EX:
jshell> "Welcome to JSHELL";
$1 ==> "Welcome to JSHELL"

jshell> $1
$1 ==> "Welcome to JSHELL"

jshell> String str = "Welcome To JSHELL";
str ==> "Welcome To JSHELL"

jshell> str
str ==> "Welcome To JSHELL"

jshell> System.out.println("Welcome To Durgasoft");
Welcome To Durgasoft

jshell> SYstem.out.println(str);
|  Error:
|  package SYstem does not exist
|  SYstem.out.println(str);
|  ^--------^

jshell> System.out.println(str);
Welcome To JSHELL

jshell> 10+20
$7 ==> 30

jshell> int a = 10;
a ==> 10

jshell> int b = 20;
b ==> 20

jshell> a*b
$10 ==> 200

jshell> System.out.println(a-b);
```

```
-10

jshell>
```

JSHELL has some default packages internally, no need to import these packages explicitly, directly we can use classes and interfaces of those packages.

If we want to know the default packages which are available in JSHELL we have to use the following command.

```
/imports
```

EX:

```
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>
```

EX:

```
jshell> List<String> list = Arrays.asList("AAA", "BBB","CCC","DDD");
list ==> [AAA, BBB, CCC, DDD]

jshell> list
list ==> [AAA, BBB, CCC, DDD]

jshell> List<String> list1 = list.stream().map(str-
>str.tolowerCase()).collect(Collectors.toL
ist());
|  Error:
|  cannot find symbol
|    symbol:   method tolowerCase()
```

```
|  List<String> list1 = list.stream().map(str-
>str.tolowerCase()).collect(Collectors.toList());
|                                              ^-------------^
|  Error:
|  incompatible types: inference variable T has incompatible bounds
|      equality constraints: java.lang.String
|      lower bounds: java.lang.Object
|  List<String> list1 = list.stream().map(str-
>str.tolowerCase()).collect(Collectors.toList());
|                       ^-------------------------------------------------
----------------^

jshell> List<String> list1 = list.stream().map(str-
>str.toLowerCase()).collect(Collectors.toL
i t());
list1 ==> [aaa, bbb, ccc, ddd]

jshell> list
list ==> [AAA, BBB, CCC, DDD]

jshell> list1
list1 ==> [aaa, bbb, ccc, ddd]

jshell>
```

In JSHELL we are able to import the packages explicitly as per the requirement.

EX:

```
jshell> import java.text.*;

jshell> NumberFormat nf = NumberFormat.getInstance(new Locale("it","IT"));
nf ==> java.text.DecimalFormat@674dc

jshell> nf.format(123456789.4567890);
$24 ==> "123.456.789,457"

jshell> Locale l = new Locale("it","IT");
l ==> it_IT

jshell> DateFormat df = DateFormat.getDateInstance(0,l);
df ==> java.text.SimpleDateFormat@96b32db5
```

```
|

jshell> df.format(new java.util.Date());
$27 ==> "sabato 5 agosto 2023"

jshell>
```

IN JSHELL, if we want to list out all the code snippets which we have provided up to now we have to use the following command on JSHELL.

/list

EX:

```
jshell> /list

   1 : "Welcome to JSHELL";
   2 : $1
   3 : String str = "Welcome To JSHELL";
   4 : str
   5 : System.out.println("Welcome To Durgasoft");
   6 : System.out.println(str);
   7 : 10+20
   8 : int a = 10;
   9 : int b = 20;
  10 : a*b
  11 : System.out.println(a-b);
  13 : List<String> list = Arrays.asList("AAA", "BBB","CCC","DDD");
  14 : list
  15 : List<String> list1 = list.stream().map(str-
>str.toLowerCase()).collect(Collectors.toList());
  16 : list
  17 : list1
  19 : drop d;
  20 : import java.sql.*;
  21 : Connection con =
DriverManager.getConnection("jdbc:odbc:nag","system","durga");
  22 : import java.text.*;
  23 : NumberFormat nf = NumberFormat.getInstance(new Locale("it","IT"));
  24 : nf.format(123456789.4567890);
  25 : Locale l = new Locale("it","IT");
  26 : DateFormat df = DateFormat.getDateInstance(0,l);
  27 : df.format(new java.util.Date());
```

In JSHELL, we are able to get the complete history of the JSHELL by using the following command.

/history

EX:

```
jshell> /history

/help intro
/help
"Welcome to JSHELL";
$1
String str = "Welcome To JSHELL";
str
System.out.println("Welcome To Durgasoft");
SYstem.out.println(str);
System.out.println(str);
10+20
int a = 10;
int b = 20;
a*b
System.out.println(a-b);
Date d = new Date();
/imports
List<String> list = Arrays.asList("AAA", "BBB","CCC","DDD");
list
List<String> list1 = list.stream().map(str-
>str.tolowerCase()).collect(Collectors.toList());
List<String> list1 = list.stream().map(str-
>str.toLowerCase()).collect(Collectors.toList());
list
list1
/help
import java.sql.*;
drop d;
import java.sql.*;
Connection con =
DriverManager.getConnection("jdbc:odbc:nag","system","durga");
import java.text.*;
NumberFormat nf = NumberFormat.getInstance(new Locale("it","IT"));
nf.format(123456789.4567890);
Locale l = new Locale("it","IT");
DateFormat df = DateFormat.getDateInstance(0,l);
```

```
df.format(new Date());
df.format(new java.util.Date());
/list
/history

jshell>
```

Variables in JSHELL:
In JSHELL , there are two types of variables.

1. Implicit variables or Scratch variables
2. Explicit Variables

Implicit variables / Scratch variables: These variables are provided by the JSHELL internally when we declare data or when we perform operations.

IN JSHELL, implicit Variables are provided in the following pattern.

```
$num
```

We are able to access the data from Implicit variables by using variable names.

EX:

```
jshell> "abc"
$1 ==> "abc"

jshell> 10+20
$2 ==> 30

jshell> "abc"+"xyz"
$3 ==> "abcxyz"

jshell> $1
$1 ==> "abc"

jshell> $2
$2 ==> 30

jshell> $3
$3 ==> "abcxyz"
```

Explicit Variables:These variables must be declared by the developers explicitly.

We can access explicit variables by using variable names directly.

EX:

```
jshell> String firstName = "Durga";
firstName ==> "Durga"

jshell> String lastName = "N";
lastName ==> "N"

jshell> int age = 22;
age ==> 22

jshell> String qual = "BTech";
qual ==> "BTech"

jshell> System.out.println(firstName+","+lastName+","+age+","+qual);
Durga,N,22,BTech

jshell> firstName
firstName ==> "Durga"

jshell> lastName
lastName ==> "N"

jshell> age
age ==> 22

jshell> qual
qual ==> "BTech"

jshell>
```

If we want to display all variables which are used in JSHELL up to now we have to use the following command.

/vars

```
jshell> /vars
|    String $1 = "abc"
|    int $2 = 30
```

```
|    String $3 = "abcxyz"
|    String firstName = "Durga"
|    String lastName = "N"
|    int age = 22
|    String qual = "BTech"

jshell>
```

If we want to get an individual variable name in JSHELL we have to use the following command.

```
/vars varName

jshell> /vars firstName
|    String firstName = "Durga"

jshell> /vars qual
|    String qual = "BTech"

jshell>
```

If we want to drop a particular variable from JSHELL we have to use the following command.

```
drop varName

jshell> drop firstName
|  replaced variable firstName, however, it cannot be referenced until class drop is declared

jshell> /vars
|    String $1 = "abc"
|    int $2 = 30
|    String $3 = "abcxyz"
|    String lastName = "N"
|    int age = 22
|    String qual = "BTech"
|    drop firstName = (not-active)

jshell>
```

Methods in JSHELL:
It is a set of instructions as a single unit representing a particular action.

In JSHELL we can declare methods without having any class declaration.

We can access the methods directly by using method names like local methods.

EX:

```
jshell> void sayHello(){
   ...>     System.out.println("Hello User!");
   ...> }
|  created method sayHello()

jshell> sayHello();
Hello User!

jshell> void welcome(String name){
   ...>     System.out.println("Hello "+name);
   ...>     System.out.println("Welcome To JSHELL");
   ...> }
|  created method welcome(String)

jshell> welcome("Durga");
Hello Durga
Welcome To JSHELL

jshell> int add(int fval, int sval){
   ...>     return fval + sval;
   ...> }
|  created method add(int,int)

jshell> add(20,5);
$22 ==> 25

jshell> int sub(int fval, int sval){
   ...>     return fval-sval;
   ...> }
|  created method sub(int,int)

jshell> sub(10,5);
$24 ==> 5
```

If we want to get all the declared methods from JSHELL then we have to use the following command.

/methods

```
jshell> /methods
|    void sayHello()
|    void welcome(String)
|    int add(int,int)
|    int sub(int,int)

jshell>
```

If we want to get a particular method details we have to use the following command.

/methods methodName

```
jshell> /methods sayHello
|    void sayHello()

jshell> /methods welcome
|    void welcome(String)

jshell> /methods add
|    int add(int,int)
```

If we want to drop a particular method from JSHELL we have to use the following command.

/drop methodName

```
jshell> /drop sayHello
|  dropped method sayHello()
|  dropped variable sayHello

jshell> /methods
|    void welcome(String)
|    int add(int,int)
|    int sub(int,int)
```

IN JSHELL, we are able to prepare methods with the same name and with the different parameters list.

```
jshell> int mul(int i, int j){
   ...>     return i*j;
   ...> }
|  created method mul(int,int)

jshell> int mul(int i, int j, int k){
   ...>     return i*j*k;
   ...> }
|  created method mul(int,int,int)

jshell> mul(10,20);
$28 ==> 200

jshell> mul(10,20,30);
$29 ==> 6000
```

Classes, Abstract classes, interfaces and enums in JSHELL:
In JSHELL , it is possible to declare classes, abstract classes, interfaces and enums.

EX:

```
jshell> class Employee{
   ...>     int eno;
   ...>     String ename;
   ...>     float esal;
   ...>     String eaddr;
   ...>     Employee(int eno, String ename, float esal, String eaddr){
   ...>         this.eno = eno;
   ...>         this.ename = ename;
   ...>         this.esal = esal;
   ...>         this.eaddr = eaddr;
   ...>     }
   ...>     public void getEmployeeDetails(){
   ...>         System.out.println("Employee Number         : "+eno);
   ...>         System.out.println("Employee Name           : "+ename);
   ...>         System.out.println("Employee Salary         : "+esal);
   ...>         System.out.println("Employee Address        : "+eaddr);
   ...>     }
   ...>
```

```
            }

   ...>     }
   ...>


         }

|  created class Employee

jshell> Employee emp = new Employee(111,"AAA",5000,"Hyd");
emp ==> Employee@2530c12

jshell> emp.getEmployeeDetails();
Employee Number         : 111
Employee Name           : AAA

jshell> emp.getEmployeeDetails();
Employee Number         : 111
Employee Name           : AAA

jshell> emp.getEmployeeDetails();
Employee Number         : 111
Employee Name           : AAA

jshell>
```

IN general, JSHELL is suggestible for simple code snippets only, not suggestible for lengthy programs. If we want to prepare java applications then we have to use Editors or IDEs.

JSHELL has its own editor internally , it is the same as notepad, to open an editor in JSHELL we have to use the following command.

/edit

If we use the above command then JSHELL Editor will open with the list of data that we provided , where we can modify, delete,... finally we have to click on the “Accept” button to reflect all modifications to JSHELL.

In JSHELL, JSHELL provided editor is not good for development, it is possible to set our own editors in place of JSHELL default editor.

IN JSHELL, to set our own editor we have to use the following command.

/set editor “Editor.exe file location”.

EX:

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

jshell> /edit
```

EX:

```
jshell> /set editor "C:/IntelliJ IDEA Community Edition
2023.1.2/bin/idea64.exe"
|  Editor set to: C:/IntelliJ IDEA Community Edition 2023.1.2/bin/idea64.exe


jshell> /edit
|  created class ToyotoCar

jshell> Car car = new ToyotoCar();
car ==> ToyotoCar@4cdf35a9

jshell> car.getCarDetails();
Innova Crysta.......

jshell> System.out.println(UserStatus.AVAILABLE);
AVAILABLE

jshell> System.out.println(UserStatus.BUSY);
BUSY

jshell> System.out.println(UserStatus.IDLE);
IDLE
```

IN JSHELL, we can save the current content in a file by using the following command.

/save fileNameAndLocation

```
jshell> /save E:/abc/xyz/data.jsh
```

In JSHELL, we can get data from a particular file to JSHELL by using the following command.

/open fileNameLocation

```
jshell> /open E:/abc/xyz/student.jsh

jshell> getStudentDetails();
STudent Details.....
```

Jar Files in JSHELL:
IN JSHELL code if we want to use some third party libraries which are available in JAR files then we have to set classpath environment variable to jar file, to set classpath environment variable to jar file we have to use the following command.

/env -class-path jarFileNameAndLocation

/env -class-path E:/abc/xyz/mysql-connector-j-8.0.33.jar

EX:
E:/abc/xyz/Employee.jsh

```
import java.sql.*;
class EmployeeDao{
   public void getEmployeeDetails(){
        Connection con = null;
      try{
         Class.forName("com.mysql.cj.jdbc.Driver");
         con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/durgadb","root","roo
t");
         Statement st = con.createStatement();
         ResultSet rs = st.executeQuery("select * from emp1");
         System.out.println("ENO\tENAME\tESAL\tEADDR");
         System.out.println("----------------------------");
         while(rs.next()){
            System.out.print(rs.getInt("ENO")+"\t");
            System.out.print(rs.getString("ENAME")+"\t");
            System.out.print(rs.getFloat("ESAL")+"\t");
            System.out.print(rs.getString("EADDR")+"\n");
         }
      }catch(Exception e){
```

```
          e.printStackTrace();
      }finally{
         try{
            con.close();
         }catch(Exception e){
            e.printStackTrace();
         }
      }
   }
}
```

```
jshell> /open E:/abc/xyz/Employee.jsh
jshell>/env -class-path E:/abc/xyz/mysql-connector-j-8.0.33.jar
jshell> empDao.getEmployeeDetails();
ENO     ENAME   ESAL    EADDR
-----------------------------
111     AAA     5000.0  Hyd
222     BBB     6000.0  Hyd
333     CCC     7000.0  Hyd
444     DDD     8000.0  Hyd
555     EEE     9000.0  Hyd
```

IN JSHELL, we can provide our own messages as startup messages by using the following command.

```
jshell -v –startup FileNameLocation
```

EX:

wish.jsh

```
String wishMessage = "Good Evening Nagoor";
System.out.println(wishMessage);
```

```
C:\Users\Administrator>jshell -v --startup E:/abc/xyz/wish.jsh
Good Evening Nagoor
|  Welcome to JShell -- Version 17.0.6
|  For an introduction type: /help intro
```

JPMS:
—-----
JPMS: Java Platform Module System

In general, in Java applications, we are able to write programs by using classes and interfaces, these classes and interfaces are combined in the form of packages, these packages combined in the form of JAR files, here we will use these jar files to execute the applications.

In the above context, JAR files are providing modularization in the applications .

The above approach is able to provide the following problems.

1. Unexpected NoClassFoundError Exception.
2. Version Conflict.
3. Security Problems
4. Monolithic Arch and Larger in Size
   —----
   —----

Unexpected NoClassFoundError Exception:
—-----------------------------------
In general, we are able to create more number of jar files as per the application requirement, where all these jar files are interdependent, where to execute the application we must keep all the jar files in the “classpath”, in this context, if any jar file is missing in the classpath environment variable then JVM will provide java.lang.NoClassDefFoundError Exception.

Employee.java

```
package p1;
import p2.*;
public class Employee{
      public void getEmpDetails(){
            System.out.println("Employee Details......");
            Account account = new Account();
            account.getAccountDetails();
      }
```

```
}

Account.java
package p2;
import p3.*;
public class Account{
	public void getAccountDetails(){
		System.out.println("Account Details.....");
		Bank bank = new Bank();
		bank.getBankDetails();
	}
}

Bank.java
package p3;
public class Bank{
	public void getBankDetails(){
		System.out.println("Bank Details.....");
	}
}

D:\java6\jpms\app01>javac -d . *.java
D:\java6\jpms\app01>jar -cvf employee.jar p1
D:\java6\jpms\app01>jar -cvf account.jar p2
D:\java6\jpms\app01>jar -cvf bank.jar p3

Delete Employee.java, Account.java, Bank.java , p1, p2, p3 from the current
location.

Test.java
import p1.*;
class Test{
	public static void main(String[] args){
		Employee employee = new Employee();
		employee.getEmpDetails();
	}
}


D:\java6\jpms\app01>javac Test.java
Test.java:1: error: package p1 does not exist
import p1.*;
^
```

```
Test.java:4: error: cannot find symbol
                Employee employee = new Employee();
                ^
  symbol:   class Employee
  location: class Test
Test.java:4: error: cannot find symbol
                Employee employee = new Employee();
                                    ^
  symbol:   class Employee
  location: class Test
3 errors

D:\java6\jpms\app01>set classpath=employee.jar;

D:\java6\jpms\app01>javac Test.java

D:\java6\jpms\app01>java Test
Employee Details......
Exception in thread "main" java.lang.NoClassDefFoundError: p2/Account


D:\java6\jpms\app01>set classpath=employee.jar;account.jar;

D:\java6\jpms\app01>javac Test.java

D:\java6\jpms\app01>java Test
Employee Details......
Account Details.....
Exception in thread "main" java.lang.NoClassDefFoundError: p3/Bank

D:\java6\jpms\app01>set classpath=employee.jar;account.jar;bank.jar;

D:\java6\jpms\app01>javac Test.java

D:\java6\jpms\app01>java Test
Employee Details......
Account Details.....
Bank Details.....
```

To overcome the above NoClassDeFoundError exception in java applications, we have to use JPMS , in the case of JPMS all dependencies are checked at starting point application execution only. If any dependency does not exist then JPMS will not start application execution.

Version Conflict:
—----------

In Java applications, we may prepare a number of jar files which are depending on each other. If we want to use these jar files in our present application then we have to keep all these jar files in “classpath”.

When we set classpath to multiple jar files, there is no guarantee whether all the jar files are prepared in the same current Java version that we are using for the present java application , there we will get the “Version Conflict” problem at runtime of the application.

EX:

Employee.java

```
package p1;
import p2.*;
public class Employee{
	public void getEmpDetails(){
		System.out.println("Employee Details......");
		Account account = new Account();
		account.getAccountDetails();
	}
}
```

Account.java

```
package p2;
import p3.*;
public class Account{
	public void getAccountDetails(){
		System.out.println("Account Details.....");
		Bank bank = new Bank();
		bank.getBankDetails();
	}
}
```

Bank.java

```
package p3;
public class Bank{
	public void getBankDetails(){
		System.out.println("Bank Details.....");
	}
}
```

Test.java

```
import p1.*;
class Test{
      public static void main(String[] args){
            Employee employee = new Employee();
            employee.getEmpDetails();
      }
}
```

```
D:\java6\jpms\app01>javac -d . Bank.java

D:\java6\jpms\app01>jar -cvf bank.jar p3
added manifest
adding: p3/(in = 0) (out= 0)(stored 0%)
adding: p3/Bank.class(in = 407) (out= 280)(deflated 31%)

D:\java6\jpms\app01>set path=C:\java\jdk1.7.0_80\bin;

D:\java6\jpms\app01>javac -d . Account.java

D:\java6\jpms\app01>jar -cvf account.jar p2
added manifest
adding: p2/(in = 0) (out= 0)(stored 0%)
adding: p2/Account.class(in = 484) (out= 328)(deflated 32%)

D:\java6\jpms\app01>set path=C:\java\jdk-17\bin;

D:\java6\jpms\app01>javac -d . Employee.java

D:\java6\jpms\app01>jar -cvf employee.jar p1
added manifest
adding: p1/(in = 0) (out= 0)(stored 0%)
adding: p1/Employee.class(in = 490) (out= 333)(deflated 32%)

D:\java6\jpms\app01>set path=C:\java\jdk1.8.0_202\bin;

D:\java6\jpms\app01>set classpath=employee.jar;account.jar;bank.jar;


D:\java6\jpms\app01>javac Test.java
Test.java:4: error: cannot access Employee
                Employee employee = new Employee();
```

```
                ^
  bad class file: employee.jar(p1/Employee.class)
    class file has wrong version 61.0, should be 52.0
    Please remove or make sure it appears in the correct subdirectory of the
classpath.
1 error

D:\java6\jpms\app01>
```

Security Problems:
In Java applications, we are able to use multiple jar files, we are able to set them in the “classpath” environment variable, in this context, if we set any JAR file in the classpath environment variable then we are able to use all packages which are available in jar file, there is no chance to hide some of the package, here JAR files are not providing Security to the packages.

JPMS is able to provide security to the packages, because JPMS is able to export the required packages to the other modules and it is able to hide some other packages to the other modules. So JPMS is able to provide security for the applications.

Monolithic Arch and Larger in Size
—---------------------------------
IN general, Java is following Monolithic Arch, that is all the packages are available in a single jar file , to execute any simple java program and if we set the jar file in the classpath environment variable then all the packages which are available in the jar file are loaded with or without the requirement, it will increase more loading time, it will reduce application performance.

In the above context, JPMS is able to provide a solution , it is able to load only the required packages, it is unable to load unnecessary packages.

In JPMS , we will create and use Modules in place of JAR files.

Module: Module is a folder or a package with the module configuration file, where module configuration file is module-info.java , it is able to provide module information like the packages names which we are using from other modules and the packages names which we want to expose to the other modules.

Steps to prepare First JPMS application:
—--
1. Create application directory Structure.
2. Create the Java files as per the application requirement.
3. Create Module Configuration file.
4. Compile the module.
5. Execute the module.

Create application directory Structure:
D:\java6\jpms
app01
|------src
| |-----moduleA
| | |------pack1
| | | |----Test.java
| | |------module-info.java

Create the Java files as per the application requirement.
Test.java
```
package pack1;
public class Test{
      public static void main(String[] args){
            System.out.println("Welcome To JPMS Programming.....");
      }
}
```

Create Module Configuration file:
module-info.java
```
module moduleA{
}
```

Compile the module:
To compile a module we have to use the following javac command.

javac –module-source-path srcFolder -d OutputFolder -m moduleName

EX:

D:\java6\jpms\app01>javac --module-source-path src -d out -m moduleA

If we compile the module by using the above command then the Compiler will create the output folder like below.

```
D:\java6\app01
Out
|------moduleA
         |------pack1
         |        |------Test.class
         |------module-info.class
```

Execute the application:
To execute a Java application which has a module we have to use the following command.

```
java –-module-path outputFolder -m moduleName/packageName.MainClassName
EX: java –-module-path out -m moduleA/pack1.Test
```

Note: In JPMS applications, module-info.java is mandatory, if we prepare a module without module-info.java file then the compiler will raise an error like “error: module moduleA not found in module source path”.

Note: IN JPMS , it is not suggestible to provide digits/numbers in the moduleNames as suffixes. If we provide digits as suffix for module name then we are able to get the following warning message.

```
src\module1\module-info.java:1: warning: [module] module name component
module1 should avoid terminal digits
module module1{
```

In general, in JPMS applications we are able to prepare more than one module and we are able to access the packages from one module to another module.

If we want to access one module package in another module then we have to use the following attributes in the module configuration file.

1. exports
2. requires

Where ‘exports’ attribute will be used in the module configuration file of a module whose packages are exposed to the other modules in order to use.

```
module-info.java
module moduleA{
      exports pack1, pack2, pakc3;
}
```

Where ‘requires’ attribute will be used in the module configuration file of a module which we want to access the packages of some other module that exports the packages.

```
module-info.java
module moduleB{
      requires moduleA;
}
```

EX:

```
D:\java6\jpms
app02
|-----src
       | —------moduleEmp
       |            |-------com
       |            |        |---durgasoft
       |            |        |       |-------emp
       |            |        |       |        |------Employee.java
       |            |-------module-info.java
       |
       |--------moduleTest
                    |--------com
                    |          |-----durgasoft
                    |          |         |------test
                    |          |         |        |-----Test.java
                    |---------module-info.java
```

```
appo2/src/moduleEmp/com/durgasoft/emp/Employee.java
package com.durgasoft.emp;
public class Employee{

      private int eno;
      private String ename;
      private float esal;
      private String eaddr;

      public Employee(int eno, String ename, float esal, String eaddr){
```

```
            this.eno = eno;
            this.ename = ename;
            this.esal = esal;
            this.eaddr = eaddr;
      }

      public void getEmpDetails(){
            System.out.println("Employee Details");
            System.out.println("-------------------------");
            System.out.println("Employee Number    : "+eno);
            System.out.println("Employee Name      : "+ename);
            System.out.println("Employee Salary    : "+esal);
            System.out.println("Employee Address   : "+eaddr);
      }
}
```

app02/src/moduleEmp/module-info.java

```
module moduleEmp{
      exports com.durgasoft.emp;
}
```

app02/src/moduleTest/com/durgasoft/test/Test.java

```
package com.durgasoft.test;
import com.durgasoft.emp.*;
public class Test{
      public static void main(String[] args){
            Employee emp = new Employee(111, "Durga", 5000, "Hyd");
            emp.getEmpDetails();
      }
}
```

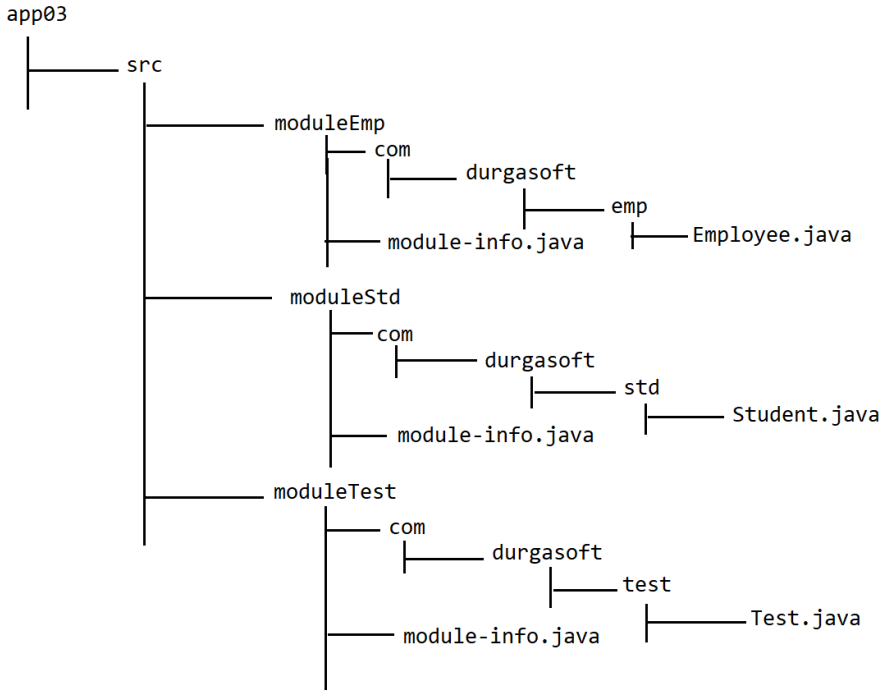
module-info.java

```
module moduleTest{
      requires moduleEmp;
}
```

```
D:\java6\jpms\app02>javac --module-source-path src -d out -m
moduleEmp,moduleTest

D:\java6\jpms\app02>java --module-path out -m
moduleTest/com.durgasoft.test.Test
Employee Details
```

```
-------------------------
Employee Number    : 111
Employee Name      : Durga
Employee Salary    : 5000.0
Employee Address   : Hyd

D:\java6\jpms\app02>
```

EX:



moduleEmp Elements:

Employee.java

```
package com.durgasoft.emp;
public class Employee{

      private int eno;
      private String ename;
      private float esal;
      private String eaddr;

      public Employee(int eno, String ename, float esal, String eaddr){
            this.eno = eno;
            this.ename = ename;
            this.esal = esal;
```

```
		this.eaddr = eaddr;
	}

	public void getEmpDetails(){
		System.out.println("Employee Details");
		System.out.println("------------------------");
		System.out.println("Employee Number    : "+eno);
		System.out.println("Employee Name      : "+ename);
		System.out.println("Employee Salary    : "+esal);
		System.out.println("Employee Address   : "+eaddr);
	}
}

module-info.java
module moduleEmp{
	exports com.durgasoft.emp;
}

moduleStd Elements:
Student.java
package com.durgasoft.std;
public class Student{

	private String sid;
	private String sname;
	private String saddr;

	public Student(String sid, String sname, String saddr){
		this.sid = sid;
		this.sname = sname;
		this.saddr = saddr;
	}

	public void getStudentDetails(){
		System.out.println("Student Details");
		System.out.println("-----------------------");
		System.out.println("Student Id      : "+sid);
		System.out.println("Student Name    : "+sname);
		System.out.println("Student Address : "+saddr);
	}
}
```

```
module-info.java
module moduleStd{
	exports com.durgasoft.std;
}

moduleTest elements:
Test.java
package com.durgasoft.test;
import com.durgasoft.std.*;
import com.durgasoft.emp.*;
public class Test{
	public static void main(String[] args){
		Student std = new Student("S-111", "Durga", "Hyd");
		std.getStudentDetails();
		System.out.println();

		Employee emp = new Employee(111,"Durga", 50000, "Hyd");
		emp.getEmpDetails();

	}
}


module-info.java
module moduleTest{
	requires moduleEmp;
	requires moduleStd;
}



D:\java6\jpms\app03>javac --module-source-path src -d out -m
moduleEmp,moduleStd,moduleTest

D:\java6\jpms\app03>java --module-path out -m
moduleTest/com.durgasoft.test.Test
Student Details
------------------------
Student Id      : S-111
Student Name    : Durga
Student Address : Hyd
```

```
Employee Details
------------------------
Employee Number    : 111
Employee Name      : Durga
Employee Salary    : 50000.0
Employee Address   : Hyd
```

Note: IN JPMS, a module exports its packages but another module is trying to use the exported packages without providing ‘requires moduleName’ attribute then the compiler will raise an error like “package packageName is not visible”.

Note: In JPMS, module does not exports its packages and if we use that packages in other modules through requires attribute then the compiler will not raise any error, but JVM will raise an exception like “ java.lang.IllegalAccessError”.

Transitive Dependencies:
In general, in java applications, we are able to prepare multiple modules as per the application requirements, in this context modules are interdependent, here if one module is dependent on the another module and another module is dependent on some other module, this type of dependency is called “Transitive Dependency”.

In a Java applications, if we have modules like moduleA, moduleB, moduleC, the moduleA depends on moduleB , moduleB depends on moduleC, here moduleC must exports its packages to the moduleB, where moduleB must exports its packages to moduleA, here moduleA wants to access moduleB packages and moduleC packages as per the transitive dependency, but it is not possible directly.

moduleA Elements:
module-info.java

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

moduleB Elements:
module-info.java

```
module moduleB{
      requires moduleC;
}
```

```
moduleC Elements:
module-info.java
module moduleC{
      –---
}
```

In the above context, moduleA is able to access moduleB members, but moduleA is unable to access moduleC members directly.

In the above context, to make available moduleC to moduleA in transitive dependency we have to use the “transitive” keyword in moduleB configuration file along with requires attributes.

```
moduleA Elements:
module-info.java
module moduleA{
      requires moduleB;
}

moduleB Elements:
module-info.java
module moduleB{
      requires transitive moduleC;
}

moduleC Elements:
module-info.java
module moduleC{
      –---
}
```

In the above case, both moduleB and moduleC are available to moduleA.

EX:

```
moduleEmp Elements:
Employee.java
package pack1;
import pack2.*;
public class Employee{
      Account acc = new Account();
      public Account getAccount(){
            return acc;
```

```
	}
	public void getEmpDetails(){
		System.out.println("Employee Details.....");
	}
}

module-info.java
module moduleEmp{
	exports pack1;
	requires transitive moduleAccount;
}

moduleAccount Elements:
Account.java
package pack2;
public class Account{
	public void getAccountDetails(){
		System.out.println("Account Details.....");
	}
}

module-info.java
module moduleAccount{
	exports pack2;
}

moduleTest Elements:
Test.java
package pack3;
import pack1.*;
import pack2.*;
public class Test{
	public static void main(String[] args){
		Employee emp = new Employee();
		emp.getEmpDetails();
		Account acc = emp.getAccount();
		acc.getAccountDetails();
	}
}

module-info.java
module moduleTest{
	requires moduleEmp;
```

```
}

D:\java6\jpms\app04>javac --module-source-path src -d out -m
moduleEmp,moduleAccount,moduleTest

D:\java6\jpms\app04>java --module-path out -m moduleTest/pack3.Test
Employee Details.....
Account Details.....
```

Optional Dependencies:

—----------

In JPMS, we are able to prepare number of modules, where modules are interdependent, in some situations, we need some modules up to compilation only but we don't required that modules at runtime, here to make available a module up to compilation and not to make available at runtime then that module is called Optional module, that modules we are able to get through “static” keyword.

EX:

moduleEmp Elements:

Employee.java

```
package pack1;
import pack2.*;
public class Employee{
      Account acc = new Account();
      public void getEmpDetails(){
            System.out.println("Employee Details.....");
            acc.getAccountDetails();
      }
}
```

module-info.java

```
module moduleEmp{
      exports pack1;
      requires static moduleAccount;
}
```

moduleAccount Elements:

Account.java

```
package pack2;
public class Account{
      public void getAccountDetails(){
```

```
            System.out.println("Accxount Details.....");
      }
}

module-info.java
module moduleAccount{
      exports pack2;
}

moduleTest Elements:
Test.java
package pack3;
import pack1.*;
public class Test{
      public static void main(String[] args){
            System.out.println("main class....");
            Employee emp = new Employee();
            emp.getEmpDetails();
      }
}


module-info.java
module moduleTest{
      requires moduleEmp;

}


D:\java6\jpms\app04>javac --module-source-path src -d out -m
moduleEmp,moduleAccount,moduleTest

D:\java6\jpms\app04>java --module-path out -m moduleTest/pack3.Test
main class....
Exception in thread "main" java.lang.NoClassDefFoundError: pack2/Account
        at moduleEmp/pack1.Employee.<init>(Employee.java:4)
        at moduleTest/pack3.Test.main(Test.java:6)
Caused by: java.lang.ClassNotFoundException: pack2.Account
        at
java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(BuiltinClassLoader
.java:641)
```

```
        at
java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(ClassLoad
ers.java:188)
        at java.base/java.lang.ClassLoader.loadClass(ClassLoader.java:520)
        ... 2 more

D:\java6\jpms\app04>
```

Limiting the modules:
—---------------------
In general, in JPMS applications, if we export a package from a module then that package is available to all other modules which are available in the present application.

```
module moduleA{
      exports pack1;
}
```

In the above context, pack1 is available to every module of the present application.

In the above context, if we want to make available a package to a particular module or modules then we have to use the ‘to’ option along with exports attribute.

```
module moduleA{
      exports pack1 to moduleB, moduleC;
}
```

EX:
moduleA elements:
A.java

```
package pack1;
public class A{
      public void m1(){
            System.out.println("m1()-A-pack1-moduleA");
      }
}
```

module-info.java

```
module moduleA{
      exports pack1 to moduleB;
}
```

moduleB elements:
B.java

```
package pack2;
import pack1.*;
public class B{
	public void m2(){
		System.out.println("m2()-B-pack2-moduleB");
		A a = new A();
		a.m1();
	}
}
```

module-info.java

```
module moduleB{
	exports pack2 to moduleC;
	requires moduleA;
}
```

moduleC elements:
C.java

```
package pack3;
import pack2.*;
public class C{
	public void m3(){
		System.out.println("m3()-C-pack3-moduleC");
		B b = new B();
		b.m2();
	}
}
```

module-info.java

```
module moduleC{
	exports pack3 to moduleTest;
	requires moduleB;
}
```

moduleTest Elements:
Test.java

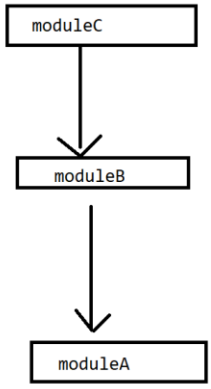
```
package test;
//import pack1.*;
//import pack2.*;
import pack3.*;
public class Test{
```

```
	public static void main(String[] args){
		/*
		A a = new A();
		a.m1();
		*/
		/*
		B b =new B();
		b.m2();
		*/
		C c = new C();
		c.m3();
	}
}

module-info.java
module moduleTest{
	//requires moduleA;
	//requires moduleB;
	requires moduleC;
}


D:\java6\jpms\app05>javac --module-source-path src -d out -m
moduleA,moduleB,moduleC,moduleTest

D:\java6\jpms\app05>java --module-path out -m moduleTest/test.Test
m3()-C-pack3-moduleC
m2()-B-pack2-moduleB
m1()-A-pack1-moduleA
```

Module Graph:
—————————
It is the Graphical representation of the modules and tier dependencies.

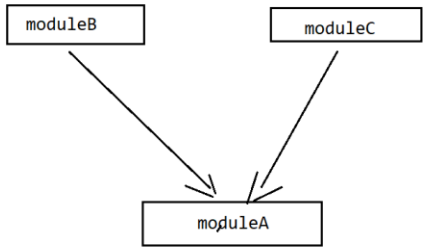
EX:

```
module moduleA{
  exports pack1 to moduleB;
}

module moduleB{
  exports pack2 to moduleC;
}

module moduleC{

}
```



EX:

```
module moduleA{
  exports pack1 to moduleB, moduleC;
}
```



Aggregator Modules:
—————————————
In general, in JPMS applications, modules may use a number of other modules, in this case if more than one module is using a set of other modules then we have to write “requires” attributes repeatedly at each and every module.

```
module moduleTest1{
      requires moduleA;
      requires moduleB;
      requires moduleC;
}



module moduleTest2{
```

```
	requires moduleA;
	requires moduleB;
	requires moduleC;

}

module moduleTest3{
	requires moduleA;
	requires moduleB;
	requires moduleC;
}
```

In the above approach, code duplication will be increased , in this context, to reduce duplicate code we will use the Aggregator module.

Aggregator module is a module which requires all other modules and it will be used in the modules which require all other modules.

```
module aggregatorModule{
	requires transitive moduleA;
	requires transitive moduleB;
	requires transitive moduleC;
}

module moduleTest1{
	requires aggregatorModule;
}
module moduleTest2{
	requires aggregatorModule;
}
module moduleTest3{
	requires aggregatorModule;
}
```

EX:
moduleA Elements:
A.java

```
package pack1;
public class A{
	public void m1(){
		System.out.println("m1()-A-pack1-moduleA");
	}
}
```

```
module-info.java
module moduleA{
	exports pack1;
}

moduleB Elements:
B.java
package pack2;
public class B{
	public void m2(){
		System.out.println("m2()-B-pack2-moduleB");

	}
}

module-info.java
module moduleB{
	exports pack2;
}

moduleC Elements:
C.java
package pack3;
public class C{
	public void m3(){
		System.out.println("m3()-C-pack3-moduleC");
	}
}

module-info.java
module moduleC{
	exports pack3;
}

moduleTest Elements:
Test.java
package test;
import pack1.*;
import pack2.*;
import pack3.*;
public class Test{
	public static void main(String[] args){
```

```
            A a = new A();
            a.m1();


            B b =new B();
            b.m2();

            C c = new C();
            c.m3();
      }
}

module-info.java
module moduleTest{
      //requires moduleA;
      //requires moduleB;
      //requires moduleC;
      requires aggregatorModule;
}

moduleTest Elements:
Test1.java
package test1;
import pack1.*;
import pack2.*;
import pack3.*;
public class Test1{
      public static void main(String[] args){
            System.out.println("main()-Test1-test1-moduleTest1");
            A a = new A();
            a.m1();

            B b = new B();
            b.m2();

            C c = new C();
            c.m3();
      }
}

module-info.java
module moduleTest1{
      //requires moduleA;
```

```
      //requires moduleB;
      //requires moduleC;
      requires aggregatorModule;
}

aggregatorModule elements:
module-info.java
module aggregatorModule{
      requires transitive moduleA;
      requires transitive moduleB;
      requires transitive moduleC;
}


D:\java6\jpms\app05>javac --module-source-path src -d out -m
moduleA,moduleB,moduleC,moduleTest,moduleTest1,aggregatorModule

D:\java6\jpms\app05>java --module-path out -m moduleTest1/test1.Test1
main()-Test1-test1-moduleTest1
m1()-A-pack1-moduleA
m2()-B-pack2-moduleB
m3()-C-pack3-moduleC

D:\java6\jpms\app05>java --module-path out -m moduleTest/test.Test
m1()-A-pack1-moduleA
m2()-B-pack2-moduleB
m3()-C-pack3-moduleC

D:\java6\jpms\app05>
```

10. Enhancements in Stream API
11. Process API Updations
12. JLinker