++

INTRODUCTION TO JPMS

++

++++++++++++++++++++++

INTRODUCTION SECTION-1

++++++++++++++++++++++

-> JPMS stands for java platform module system.

-> Introduced in the year 2017 under the version java 9.

-> Development was begun in 2005.

-> The JEP(jdk enhancement proposal) for this jpms was raised in 2005.

-> The project name under which jpms was developed was jigsam.

++++++++++++++++++++++

INTRODUCTION SECTION-2

++++++++++++++++++++++

-> Until the java version 8, every thing was based on jar files.

-> jar: group of packages, where in each packages contain group of .class files.

-> From the java version 9, we dont have a concept of jar. every thing is dealed with modules.

-> " Module is nothing but the group of packages with a special file called module-info.java."

-> NOTE:

-> JAR do not maintain/ include any configuration information.

-> In case of modules the configuration information is stored in a special file called "module-info.java".

-> every module should have a file called module-info.java, if the jvm does'nt find it we get an error.

-> In java 9 the best thing is that all the predefined classes are been divided into set of modules.

-> Where as in java 8, all the 4300+ classes were included in one single jar file named rt.jar.

-> example for some modules in java 9 are:

-> java.base.module

-> java.sql.module

-> java.rmi.module

->java.logging.module

->java.desktop.module

-> Let us know some of the packages that are available in some modules.

-> java.base.module

->java.lang

->java.util

->java.io

-> java.desktop.module

-> java.awt

-> java.swing

-> Now let us know how we can get the module name of a particular class in java.

-> System.out.println(String.class.getModule());

-> System.out.println(System.class.getModule());

-> So we got to know that we can get the module name of a class using the method available getModule().

-> The module-info.java contains the description of the module. The description include such as:

-> name of the module

-> dependencies

-> public packages

++++++++++++++++++++++

INTRODUCTION SECTION-3

++++++++++++++++++++++

-> In this section we shell discuss the differences between the jar and modules.

Difference - 1:

-> jar is only a group of packages

-> module is also a group of packages including a special file module-info.java

----

Difference - 2:

-> In jar files if there are any dependencies then jvm will throw no class def found
   error.

-> But here no such kind of errors, as we will specify the dependencies in
   module-info.java.

Difference - 3:

-> jar files should be placed in classpath in perticular order, else it may lead to
   version conflicts and etc..

-> module path, java considers the reqiured modules in the module-info.java, so there
   wont be any version confilcts and etc..

Difference - 4:

-> In case of jar, security is less in terms of accessing the packages
   (no restrictions in accessing the packages).

-> In modules, we can specify the access of only perticular packages required.

Difference - 5:

-> Jar based applications become heavier.

-> In module based, As we can get required modules specifically the application
   become lighter.

++++++++++++++++++++++

INTRODUCTION SECTION-4

++++++++++++++++++++++

-> Now we shell know what is JAR HELL or CLASSPATH HELL

-> JAR HELL: The problems that are associated with jar files are known as jar hell.

->Problems:

-> no class def found

-> version conflicts

-> security

-> monolithic structure(bigger size.(rt.jar))

-> Goals of JPMS:

-> Reliable configuration(module-info.java).

-> Strong encapsulation and security(access to only required packages).

-> scalable java platform(small memory multiple files).

-> performance and memory improvements.

++++++++++++++++++++++

INTRODUCTION SECTION-5

++++++++++++++++++++++

-> In this section we are going to know how to write module-info.java

-> structure of module-info.java:

```
module module_name{
 // 1. what other modules requires by this module.
 //example:


 requires moduleA;
 requires moduleB;
 .
```

```
         .
         // 2. what packages exported by this module.
         //example

         exports pack2;
         exports pack3;
         .
         .

        }
```

-> here module,requires, and exports are keywords.

++++++++++++++++++++++

INTRODUCTION SECTION-6

++++++++++++++++++++++

-> In this section we shell the steps to develop first module based application:

```
  ++++++++++++
  | moduleA  |
  ++++++++++++
       |       +++++++++++++++++++
       |++++++++| module-info.java |
       |        +++++++++++++++++++
       |             +++++++++++
       |++++++++|  pack1  |
       |              +++++++++++
               |       +++++++++++++
               |++++++++| test.java |
               |       +++++++++++++
```

-> This is how we need to create the path and locate the files. As like in any IDE.

-> For compiling the java module the command that is to be used is:

```
-----------------------------------------------------

 -> javac --module-source-path src -d out -m module_name

-----------------------------------------------------
```

-> The out folder will contain the .class files of the src folder files.

-> For running the java module the command that is to be used is:

```
-------------------------------------------------------------

 -> java --module-path out -m module_name/package_name.file_name

-------------------------------------------------------------
```

-> The folders of the above things with example is placed(please check for more details.)

-> Folder names:

    -> src

    -> out

    -> image of the output

-> NOTE:

    -> package statement is highly mandatory in java files.

    -> Avoid giving module name terminating with a digit. Because it will surely gonna
      throw warning but the compilation does'nt stop.

++

INTRODUCTION TO JPMS-2

++

++++++++++++++++++++++

INTRODUCTION SECTION-1

++++++++++++++++++++++

-> Here lets discuss about the cyclic dependencies in modular programming.

-> The cyclic basicaly mean, say we have two modules named moduleA and moduleB, if moduleA
   requires moduleB and moduleB requires moduleA then this is called a cyclic dependency.

```
++++++++++++++              ++++++++++++
  |  moduleA   | <----------> |   moduleB  |
 ++++++++++++++             ++++++++++++
```

-> The java9 module based programming doesn't all cyclic dependencies. it will throw an error.

-> The observable modules what we have in java9 do not have cyclic dependencies.

-> If we are developing any application, we need to make sure that we do not have any cyclic
   dependencies.

++++++++++++++++++++++

INTRODUCTION SECTION-2

++++++++++++++++++++++

-> Previously we have discussed about cyclic dependencies in java9.

-> Now we shell discuss about the transitve dependencies in java9.

-> yes, in java9 the transitive dependencies are allowed.

-> say for example we have three modules, moduleA,moduleB,and moduleC

   -> moduleB will read the moduleC

   -> moduleA reads the moduleB

   -> moduleA again requires moduleC

   -> instead we seperatly make use of moduleC in moduleA, we can go for transitive depe
     -ndency, as moduleB is already using the moduleC.

```
++++++++++++    reads     +++++++++++++    reads     +++++++++++++
| moduleA  |------------> |  moduleB  |-------------> |  moduleC  |
++++++++++++              +++++++++++++              +++++++++++++
```

-> Example program:

  -> moduleC

     -> module-info.java

        -> export pack3;

     -> pack3

```
            ->Test3.java
               package pack3;
               public class Test3{
                 public void print3(){
                   System.out.println("This is from moduleC pack3");

                 }
               }
-> moduleB
       -> module-info.java
            -> exports pack2;
            -> requires transitive moduleC;
       -> pack2
            -> Test2.java
               package pack2;
               import pack3.Test3;
               public class Test2{
                  public t3 print2(){
                     System.out.println("This is from moduleB and pack2");
                     Test3 t3 = new Test3();
                     return t3;
                  }
               }
-> moduleA
       -> module-info.java
            -> requires moduleB
       -> pack1
            -> Test.java
              package pack1;
              import pack2.Test2;
              import pack3.Test3;
```

```
class Test{
  public static void main(String[] args){
    System.out.println("This is from moduleA and pack1");
    Test2 t2 = new Test2;
    Test3 t3 = t2.print2();
    t3.print3();
  }


}
```

->NOTE:

-> In module-info.java of moduleB, if we specify

-> requires transitive moduleC

-> then which ever module requires the moduleB can also use moduleC too.

-> We must more concentrate on working with transitive dependencies in case of module based programming.

++++++++++++++++++++++

INTRODUCTION SECTION-3

++++++++++++++++++++++

-> Now we shell discuss, what if we have same package name in different modules.

-> In case of non-modular based programming,

-> say we have two jar files,

-> jar1

-> pack1

-> pack2

-> jar2

-> pack2

-> pack3

-> If this is the case then when we run, the jvm will combine the packages with same name.

-> NOTE: we will run using classpath where jar1 has class-path1 and jar2 has class-path2.

-> In the case of modular programming, say we have two modules

```
    -> moduleA
        -> pack1
        -> pack2
        -> module-info.java
    -> moduleB
        -> pack3
        -> pack2
        -> module-info.java
```

-> When we compile them, the JVM will throw an error.

-> So the package name in all the modules need to be different, else we get error.

-> The abservable modules also have distinct package names.

++++++++++++++++++++++

INTRODUCTION SECTION-4

++++++++++++++++++++++

-> In this section we shell discuss about the concept called aggrigate modules.

-> so what is a aggrigate module?

-> say for example we have four modules moduleA, moduleB, moduleC, and moduleD.

-> now moduleD requires all the other modules.

-> so it is a studious job in order to write that requires moduleA,...moduleC in module-info

.java of moduleD. Therefore what we do is that we create another module which doesn't have any

specification but only have the module-info.java, including the required modules of moduleD.

-> then we just make use of this module in other modules which all requires the modules specified in this intermediate module.

-> This intermediate module is called as aggrigate module.

++++++++++++++++++++++

INTRODUCTION SECTION-5

++++++++++++++++++++++

-> In this section we will be discussing about the qualified exports

-> This is actually based on security purpose

-> say we have a moduleA which requires only perticular package of moduleB, then

this is been done using a concept of qualified exports. i.e

```
exports package_name to module_name,module_name....;
exports package_name to module_name;
```

++

INTRODUCTION PART - 3

++

++++++++++++++++++++++

INTRODUCTION SECTION-1

++++++++++++++++++++++

-> There are mainly two goals of JPMS

-> Divide and conqure

-> Encapsulation

-> Divide and conqure:

-> There will be a big problem

-> Dividing it into sub-problems

-> Finding solution to those

-> Combine the solution

-> Its like we have rt.jar which is huge, where in we divide the .class files into some

modules in case of module based programming i.e JPMS, and using module-info.java we can combine

differnt modules.

-> Encapsulation:

-> It basically mean that we are going to encapsulate certain modules under the name of modules

-> So that the accessing becomes more secured.

-> This is given by:

```
            ++++++++++++          +++++++++++++
            | moduleA  | -----------> |  moduleB  |
            ++++++++++++      |      +++++++++++++
                              |
                              |      +++++++++++++
                              |-----> | moduleC   |
                                     +++++++++++++
```

-> As per the above diagram the moduleA has some classes,but some of them requires the methodes

or other data from moduleB and moduleC. Which says the encapsulated classe from one module is secured

and can be accessed specifically.

-> One more advantage is independency among the modules i.e if there exsist a error in moduleA

only that will be affected not others.

++++++++++++++++++++++

INTRODUCTION SECTION-2

++++++++++++++++++++++

-> Creating more than one modules and interating between them.

-> Let us know how to create the path of this, for working on two modules.

```
  +++++++
  | src |
  +++++++
     |     +++++++++++
     |------| moduleA |
     |     +++++++++++
     |           |    ++++++++++++
     |           |-----| package1 |
     |           |    ++++++++++++
     |           |           |    ++++++++++++++
     |              |              |-----| .java file |
     |              |                   ++++++++++++++
     |              |     ++++++++++++++++++++
     |              |-----| module-info.java |
     |                   ++++++++++++++++++++
     |     +++++++++++
     |------| moduleB |
           +++++++++++
                  |     ++++++++++++
                  |-----| package2 |
                  |     ++++++++++++
                  |              |     ++++++++++++++
                  |              |-----| .java file |
                  |                   ++++++++++++++
                  |     ++++++++++++++++++++
                  |-----| module-info.java |
                       ++++++++++++++++++++
```

-> So as per the above directory stucture specified we need to create the files and folders

-> say for example moduleA requires the method inside the moduleB then the program needs to be written as

-> moduleA

```
      -> module-info.java
            module moduleA{
            requires moduleB;
           }
      -> package1
          -> Test.java
              package package1;
              import package2.Test2;
              public class Test{
              public static void main(String[] args){
                Test2 t2 = new Test2();

                t2.print();

              }


            }
```

-> moduleB

```
    -> module-info.java

        module moduleB{

          exports package2;

        }

    -> package2

        -> Test2.java

            package package2;

            public class Test2{

            public void print(){

                System.out.println("This statement is from moduleB and package2");

            }

            }
```

-> After all these how we shell compile and run:

-> For compiling:

```
==============================================================
 javac --module-source-path src -d out -m moduleA,moduleB
==============================================================
```

-> After when you execute the above command the directory "out" with similar path as "src" will be created, but all the files in "out" will have only .class files.

-> For running we use the command:

```
====================================================
 java --module-path out -m moduleA/package1.Test
====================================================
```

-> NOTE:

-> If we dont use exports keyword inside the module-info.java then the method print()

wont be available/ accessable by moduleA

-> If we dont use requires in module-info.java the same this is applied.

-> (*)If we try to export a perticular class or a method from the package we will get an error. So the JVM will treat the word specified after the exports keyword as a package name only.

-> If we are using two different out files after compiling we have to use different command to run:

```
=================================================================
 java --upgrade-module-path out;out1 -m moduleA/package1.Test
=================================================================
```

-> Similarly say if we have maintained two seperate src folders then how shell we compile it.

-> Using the following command:

```
=================================================================
 javac --module-source-path src;src2 -d out -m moduleA,moduleB
=================================================================
```

++++++++++++++++++++++

INTRODUCTION SECTION-3

++++++++++++++++++++++

-> In this section we shell discuss discuss how to works with three modules.

-> say we have three modules

- -> moduleA
- -> moduleB
- -> moduleC

-> moduleA requires a method in moduleB

-> moduleB exports the package2

-> moduleC requires moduleA

-> Now if i delete moduleB then automatically the JVM will throw the error without even executing furthur lines. But in java8 it will executing the furthur line of code.

-> Now let us take one scenario, where in i will be deleting the "src2" folder of moduleB.

-> And the rest two modules moduleA and moduleB are in "src" folder, but i have the compiled "out2" folder of the

deleted "src2" folder.

-> for this case how do we compile:

-> the compilation code is given below:

```
==============================================================================
 javac --module-source-path src --module-path out2 -d out -m moduleA,moduleC
==============================================================================
```

-> while running we have to use the command:

```
========================================================
 java --module-path out;out2 -m moduleC/package3.Test3
========================================================
```

++++++++++++++++++++++

INTRODUCTION SECTION-4

++++++++++++++++++++++

-> We all know that java9 contains some ready-made modules

-> so how can we access those

-> See in this case of readymade modules we need not have to specify the --module-path option

it will be directly been mapped to java command itself.

-> We have another concept called observable modules:

->observable modules:

-> built in modules

-> compile module-path is already provided to java command

-> If we want to know which all are the built in modules that are available then we use the option

```
===============
--list-modules
===============
```

-> This option should be used with "java" command i.e for example

```
=====================
 java --list-modules
=====================
```

-> if you want to know the user defined modules you need to specify the module-path i.e

```
=======================================
 java --module-path out --list-modules
=======================================
```

-> where out is the complied src folder.

++++++++++++++++++++++

INTRODUCTION SECTION-5

++++++++++++++++++++++

-> Readability and accessability

-> moduleA can read moduleB as it has requires specified in module-info.java

-> moduleB cannot read moduleA as it has specified only the exports in module-info.java

-> Class m2 can be accessed by moduleA as it is made public.

-> Class m3 cannot be accessed as it is made default.

-> Classes m4 and m5 cannt be accesssed at all, as moduleB doesn't export package3.

-> By using this we can actually make or impliment the security.