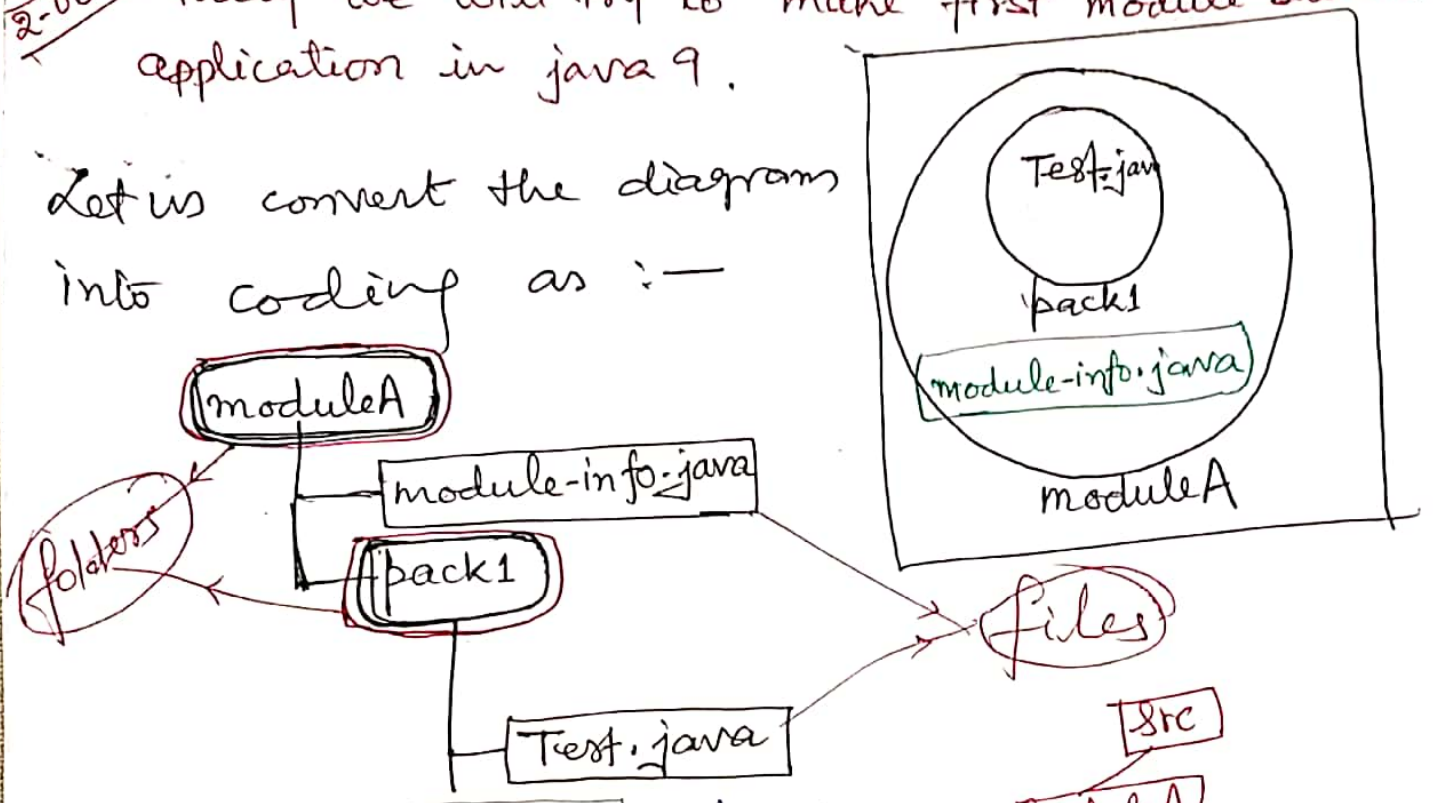
2-06-2020

Today we will try to make first module based application in java 9.

Let us convert the diagram into coding as :—



module-info.java Coding

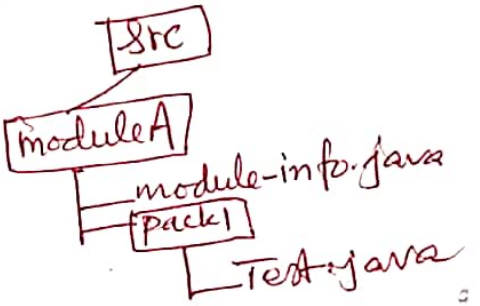
```
module moduleA
{
}
```

Very Important

In modular java programming — every class should be compulsorily belongs to a package.

Test.java

```
package pack1;
public class Test
{
public static void main(String [] args)
  {
  System.out.println("First Module Application");
  }
}
```



Source folder

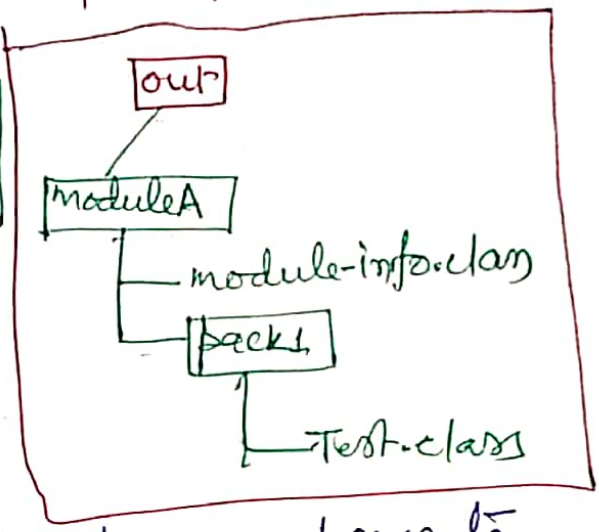
COMPILE and RUN :—

```
javac --module-source-path src -d out -m moduleA
java --module-path out -m moduleA/pack1.Test
```

After compilation — our directory 'out' structure is like as :—

```
javac --module-source-path src -d
  out -m moduleA ↵
```

```
java --module-path out -m
  moduleA/pack1.Test ↵
```



Note :- In our current working directory, we have to

→ create a folder (named here — src)
→ Then, Goto-src ; create a folder moduleA ;
→ Then goto — moduleA ; create a folder — pack1 ;
→ Come back to your current working directory.
(Back to src folder or parent of src folder)
→ Create Test.java and save it in pack1 folder
→ Create module-info.java and save it in moduleA folder
→ Compile Now as giving the command :—

```
javac --module-source-path src -d out -m moduleA ↵
```

- javac — Command
- --module-source-path src — to specify module's parent directory i.e., path of module directory.
- -d out — to specify directory name for output class files for entire module application.
- -m moduleA — to specify name of the module being created by us.

javac command has three different arguments here —

1. --module-source-path src
2. -d out
3. -m moduleA

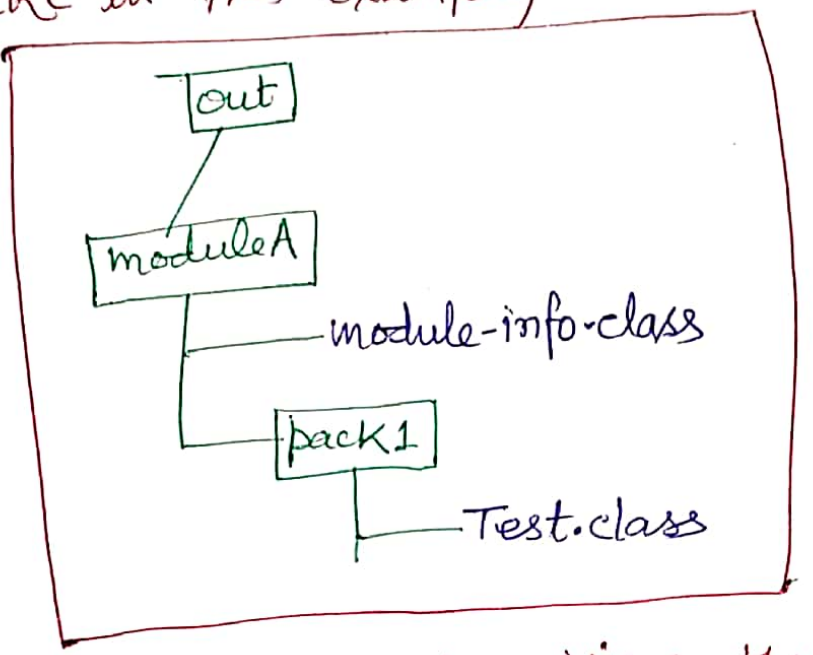
(1), (2), (3) — Options ; src, out, moduleA — Values

Each option is followed by its values

→ option we have to remember

→ Values according to specified name (need).

After successful Compilation we have the following structure (like in this example):—



You can see the structure by giving the following commands from your current working directory (Java home directory):—

```
tree out /f ↵
```

You can also see your created source module directory structure as:—

```
tree moduleA /f ↵
```

To run the created module :—

```
java --module-path out -m moduleA/pack1.Test ↵
```

- java → Command
- --module-path out → to specify path of module
- -m moduleA/pack1.Test → to specify fully specified path for class file being created for output.

Java command has two arguments:—
① --module-path out
② -m moduleA/pack1.Test

Important to remember :—

1. Every module has module-info.java file compulsorily contained. (mandatory)
2. Every class's should be declared inside a package only. That means, every class coding starts with package <package-name>;
   That means unnamed package is not allowed for any class, enum or interface.
3. Module name is not recommended to terminate with digits that means it is not recommended by the compiler - compiler gives warning.
   That means — we should rename our module as — moduleA ✓
   but not recommended — module1 ✗

Now, we will going to learn various possible ways to compile a module :—

First way we have learned in above (previous) program
Other ways are as follows :—

① javac --module-source-path src -d out (-m) moduleA

② javac --module-source-path src -d out (--module) moduleA

Compile directly java files only :- (rarely used)

③ javac --module-source-path src -d out src/moduleA/module-info.java src/moduleA/pack1/Test.java ↵

④ Complete path also can be given as :—

javac --module-source-path src -d out D:/Java/src/moduleA/module-info.java D:/Java/src/moduleA/pack1/Test.java ↵

Hence, there are 4 different ways to compile java source files using module based application development.

Now, so far we have learn how to compile and Run java module based application in java 9, with various different ways.

Optional Dependency →

If we have Modules like :-

```
module moduleA
{
}
```

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

```
module moduleC
{
}
```

Optional dependency

optional dependency requires Directive with static keyword :-

→ If a module has requires moduleA; statement inside, then it is mandatory to present moduleA at compile time.

→ If a module has requires static moduleC; statement inside, then it is ~~optional~~ mandatory that moduleC is present at compile time, but optional at runtime.

Such type of dependency is called optional dependency.

Hence :-

Optional dependency — Mandatory at Compile time but optional at Run-time.

```
requires static module_name;
```

Mandatory dependency — Mandatory at both Compile time and at Run-time.

```
requires module_name;
```

Cyclic Dependency :—

```
class A extends B
{
}
```

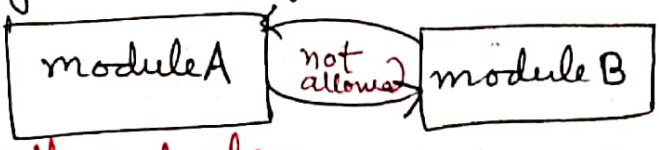
A → child class, B → Base class


```
class B extends A
{
}
```

B → child, A → Base

* cyclic inheritance is not allowed in java

Similarly in case of modules also :—



not allowed also.

Eg:-

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

module-info.java
moduleA

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

module-info.java
moduleB

Cyclic dependency is not allowed between modules also.

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.

Example:-

moduleA components :-

module-info.java

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

moduleB components :—

module-info.java

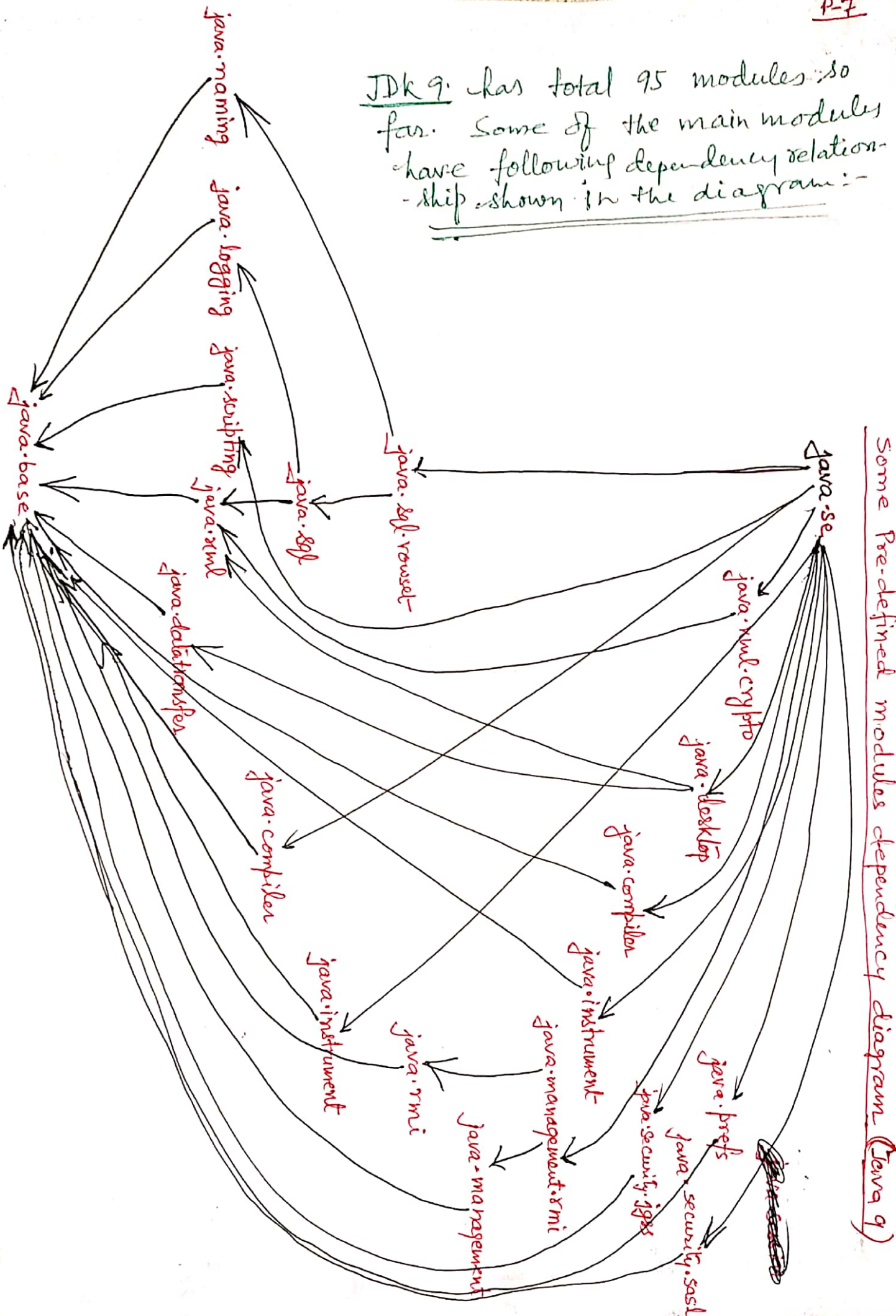
```
module moduleB
{
 requires moduleA;
}
```

If we compile this code :— immediately we will get cyclic dependency involving moduleA ⇒ requires moduleA;

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

There may be a chance of cyclic dependency between more than two modules also — moduleA requires moduleB; moduleB requires moduleC; moduleC requires moduleA; — It is also not allowed.

JDK 9 has total 95 modules so far. Some of the main modules have following dependency relationship shown in the diagram :-



Some Pre-defined modules dependency diagram (Java 9)

exports qualifier

Today, we will try to demonstrate exports qualifier in module programming.

Suppose we have moduleA having package pack1 and pack2.

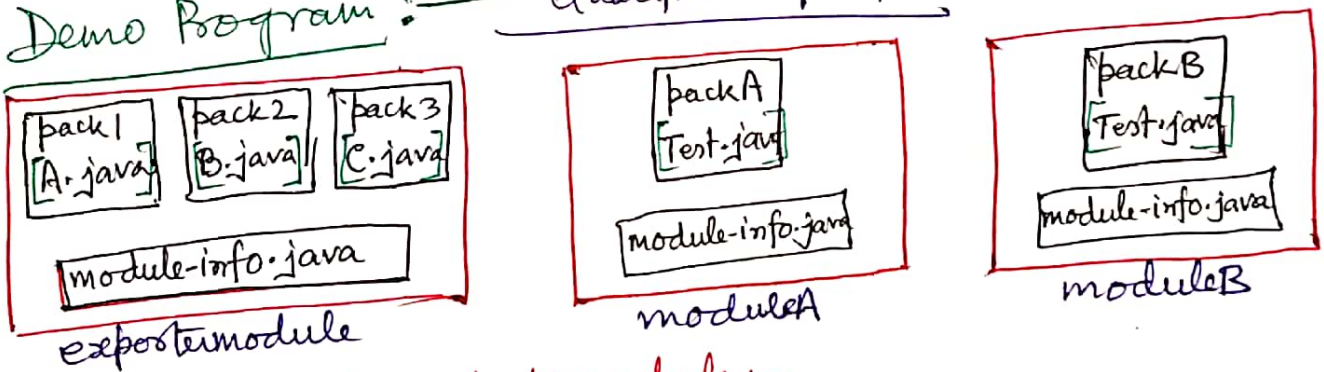
exports

```
module moduleA
{
exports pack1; // pack1 is available for any external module
exports pack1 to moduleB; // pack1 is only available to moduleB
exports pack1 to moduleB, moduleC; // pack1 is available for moduleB, moduleC.
}
```

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>, ...;

Demo Program:- Qualified Exports



Components of exportermodule:-

A.java

```
package pack1;
public class A
{
}
```

B.java

```
package pack2;
public class B
{
}
```

C.java

```
package pack3;
public class C
{
}
```

module-info.java

```
module exportermodule
{
    exports pack1;
    exports pack2 to moduleA;
    exports pack3 to moduleA, moduleB;
}
```

	moduleA	moduleB
pack1	✓	✓
pack2	✓	✗
pack3	✓	✓

Components of moduleA:-

Test.java

```
package packA;
    import pack1.A;
    import pack2.B;
    import pack3.C;
public class Test {
  public static void main(String [] args)
    {
System.out.println("Qualified Export demo");
    }
}
```

module-info.java

```
module moduleA
{ requires exportermodule;
}
```

Components of moduleB:-

Test.java

```
package packB;
import pack1.A; //Here we cannot import pack2.B
import pack3.C;
public class Test {
    public static void main(String [] args) {

    }
}
```

module-info.java

```
module moduleB
{
 requires exportermodule;
}
```

Compile :-

javac --module-source-path src -d out -m exporteemodule, moduleA ↵

Run :-

java --module-path out -m moduleA/packA.Test ↵

Similarly, compile -

javac --module-source-path src -d out -m exportermodule, moduleB ↵

Run :-

java --module-path out -m moduleB/packB.Test ↵

Successfully Runs. because we have not imported pack2.B in moduleB (because in exporterimodule, there is no exports qualifier given for pack2 to moduleB.).

Some memorable Facts — (Valid statements)

requires moduleA; → Valid statement
requires static moduleA; → Valid
requires transitive moduleA; → Valid
exports pack1; — Valid.
exports pack1 to moduleA, moduleB; — Valid
exports pack1 to moduleA; — Valid.

END