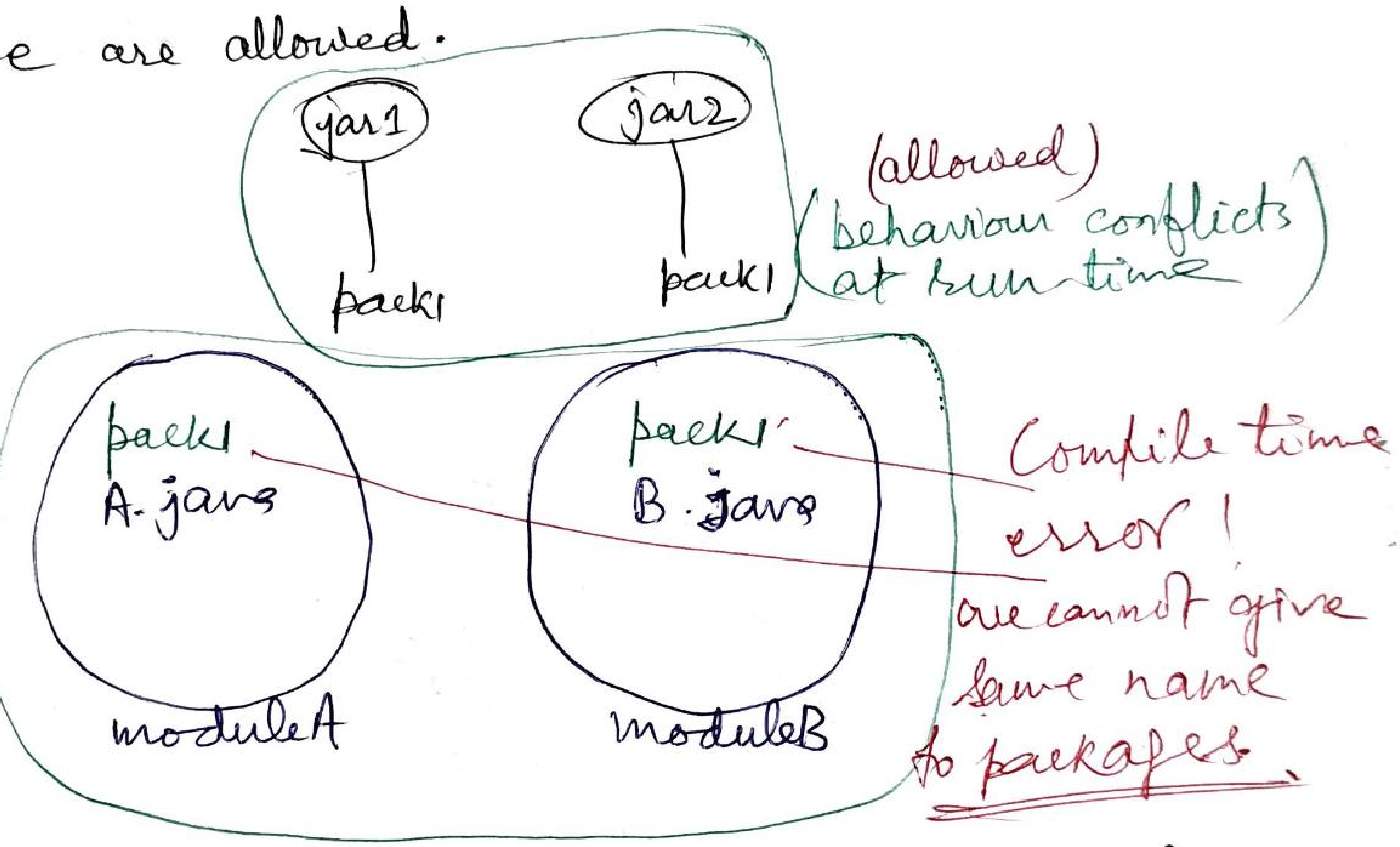
4-06-2020

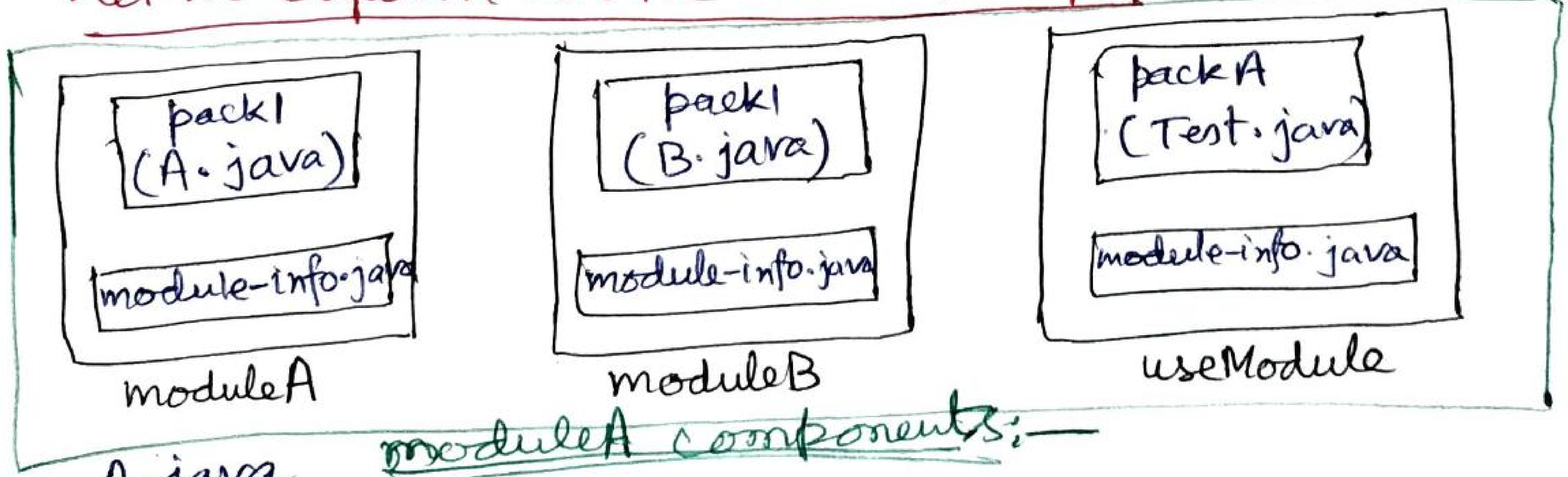
Today we will discuss about another small topic – package naming conflicts

Sometimes in older version of java – we may face two jar files having same name of package inside them. Then there should be package naming conflicts and version conflicts.

Since in jar files having same package name are allowed.



Let us explain with an example :— Demo Program:—



moduleA components:—

A.java

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

module-info.java

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

moduleB components:-

B.java

```
package pack1; // package pack2;
public class B {
   public void m1() {
          System.out.println("moduleB method");
                  }
        }
```

module-info.java

```
module moduleB
{
  exports pack1; // exports pack2;
}
```

useModule components :-

Test.java

```
package packA;
public class Test {
    public static void main(String [] args)
     {
  System.out.println("Package Naming Conflicts");
     }
  }
```

module-info.java

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

Compile :-

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

It will give error :- while compiling.

Hence in module based approach - java package naming conflicts are not available.

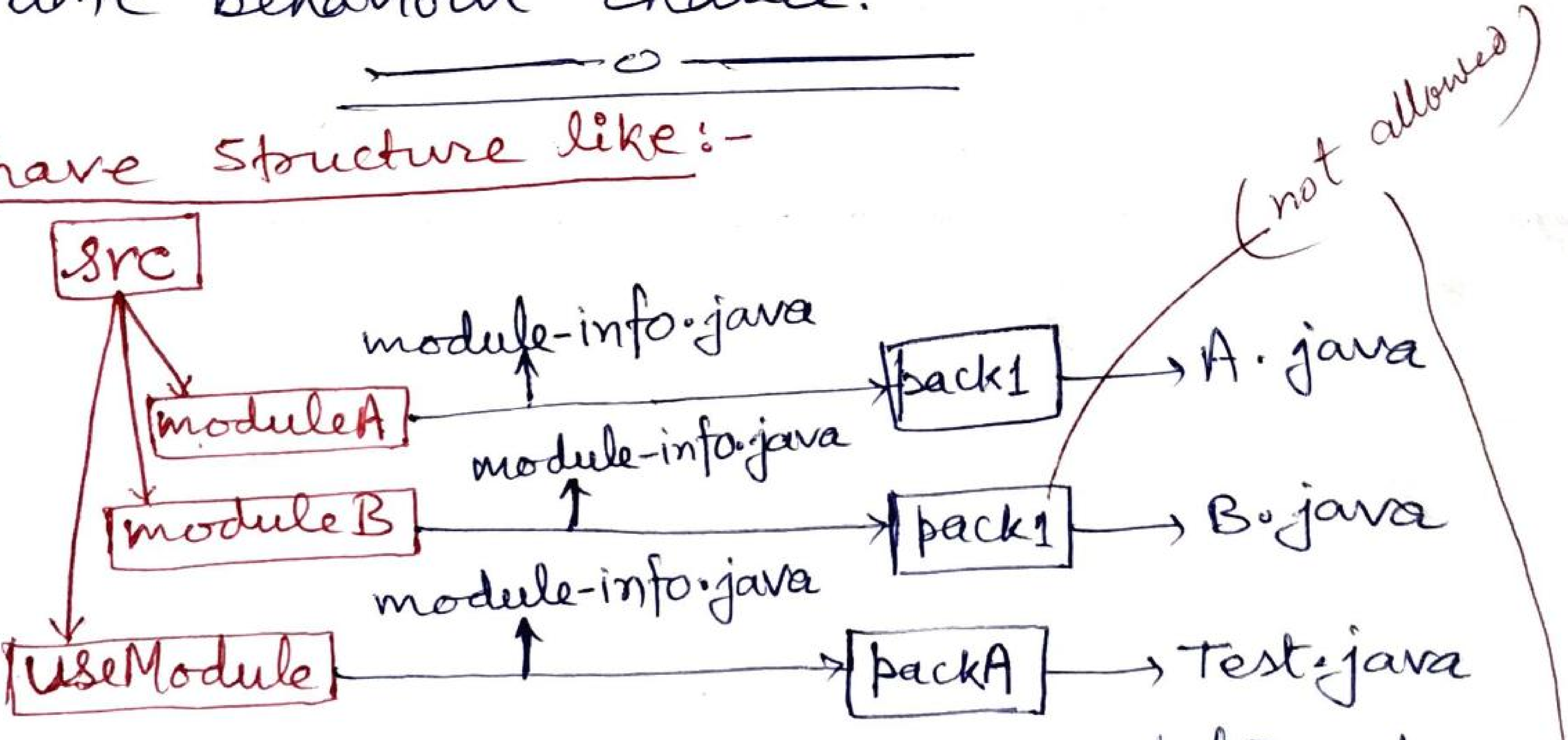
Since, we cannot allowed to give same package name in our modules.
Modify it & Run again - it will successful.

$\boxed{\text{Conclusion}}$ → Two jar files can contain a package with the same name which may cause version conflicts and abnormal behaviour at the run-time

If Two module contain package with the same name then compiler would not accept it.

- In modular programming there is no abnormal run time behaviour chance.

We have structure like :-



① If we modify — pack1 with pack2 in moduleB, code will successfully compile — after the three modifications
② Again modify — $\boxed{\text{package pack2;}}$ inside B.java
③ Third modify — moduleB → module-info.java → exports pack2;

Module Resolution Process → JVM performs module resolution process at the time of compiling. We can see the module resolution process by the command at run-time as:-

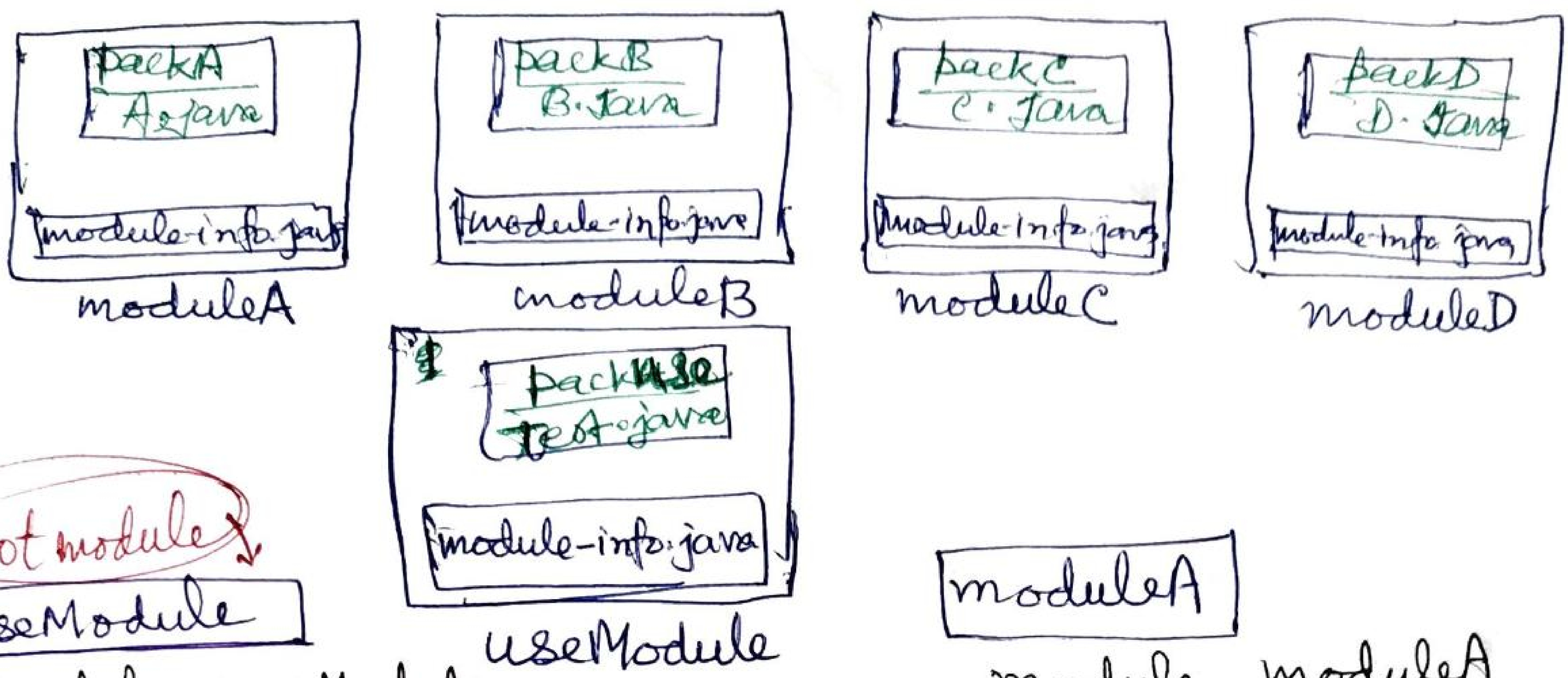
Running Command :-

```
java --module-path out -m useModule/packA.Test↵
java --module-path out --show-module-resolution
                        -m useModule/packA.Test↵
```

In traditional classpath, JVM would not check the required .class files at the beginning. While executing program if JVM requires 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 ~~class~~ 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 execution.

Let us try to develop following module based structure — whose diagram is as follows:— src3



Root module

useModule
```
module useModule
{
  requires moduleA;
}
```

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

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

moduleC
```
module moduleC
{
}
```

moduleD
```
module moduleD
{
}
```

moduleC/packC/C.java
```
package packC;
public class C{

}
```

moduleD/packD/D.java
```
package packD;
public class D {

}
```

useModule/packuse/Test.java
```
package packuse;
public class Test
{
  public static void main(String [] args)
  {
   System.out.println(" Module Resolution Process Demo !");
  }
}
```

moduleA/packA/A.java
```
package packA;
public class A{
}
```

moduleB/packB/B.java
```
package packB;
public class B{
}
```

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 is —

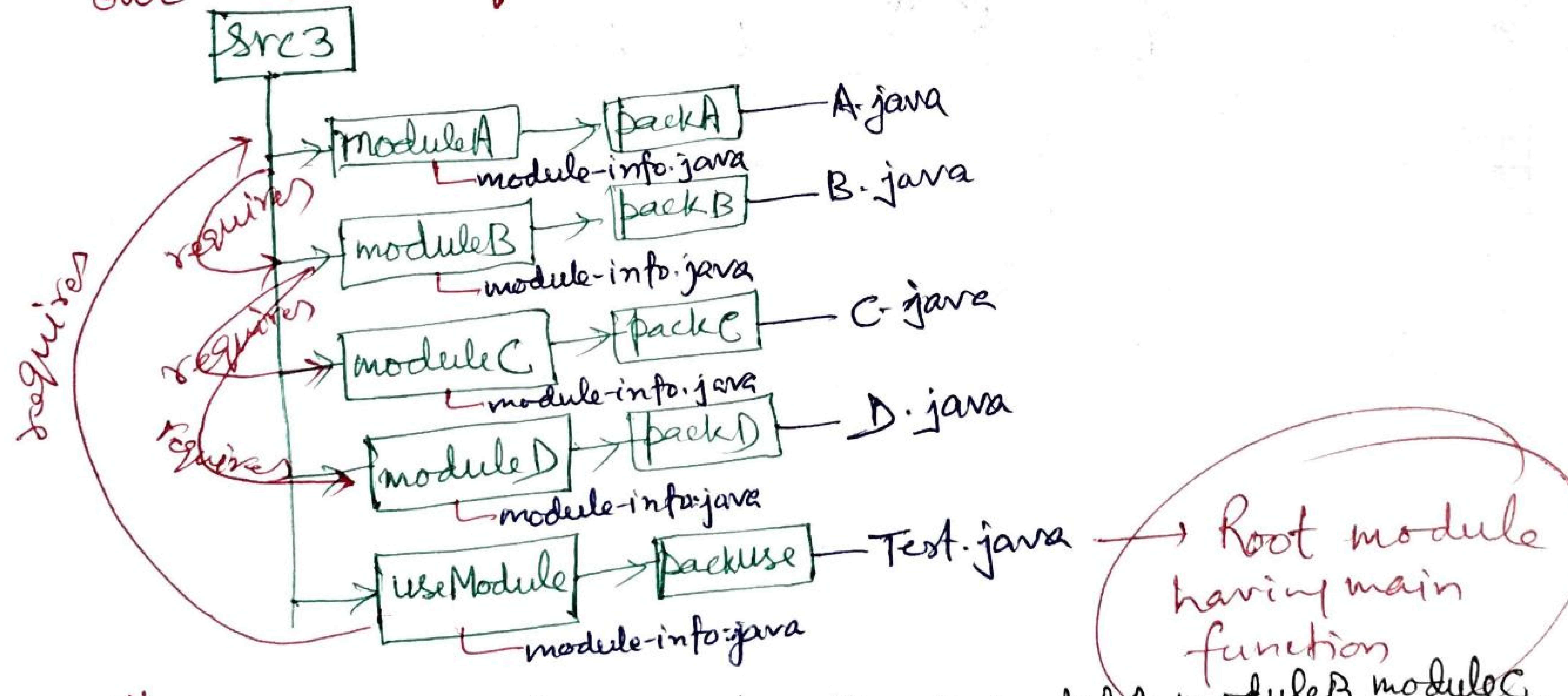
1. We will get errors if any dependent module is not available.
2. We will get error if multiple modules with the same name.
3. We will get errors if any cyclic dependency.
4. We will get errors if two modules contain packages with the same name.

Note:- Keywords of Java-9 :—

module, requires, transitive, exports; — Restricted for use in module-info.java

⟹ In normal java program no restrictions and we can use for identifier purpose also.

Our Structure of module :—



Compile:-
```
javac --module-source-path src3 -d out3 -m moduleA,moduleB,moduleC,
                                     moduleD, useModule ↵
```

~~javac~~
Run:-
```
java --module-path out3 -m useModule/packuse.Test ↵
```
or
```
java --module-path out3 --show-module-resolution -m
                                  useModule/packuse.Test ↵
```
To view the resolution process details.

— END —