

Java 9 Notes

➤ Jshell

- ◆ Jshell is nothing but Java REPL tool.
- ◆ REPL : Read, Evaluate, print, Loop (Repeat the same like Read, Evaluate and print)
- ◆ Jshell is known as Interactive console

What is Jshell ?

- ◆ The Java Shell tool (JShell) is an interactive tool for learning the Java programming language and prototyping Java code. JShell is a Read-Evaluate-Print Loop (REPL), which evaluates declarations, statements, and expressions as they are entered and immediately shows the results. The tool is run from the command line
- ◆ For Beginners -> Learning and syntax
- ◆ For Developers: -> Execute part of program in simple way
- ◆ Jshell is not meant for main coding
- ◆ It is helpful for testing part of program snippets

How to open Jshell ?

- ◆ To open Jshell go to command prompt and type jshell.
If you don't have the Java 9 installed on your machine system will throw the error
SO install the Java 9 and change the class path and JAVA_HOME variable paths in environment variables to point to java9
- ◆ Now close the previous cmd window and open new cmd window and type Jshell now you can see the Jshell window

Commands of Jshell

Jshell - To Open Jshell from command prompt
Jshell -v : Open Jshell in verbose mode
/exit : To exit from the Jshell

Advantages of Jshell

- ◆ Jshell has reduced all the efforts that are required to run a Java program and test a business logic. If we don't use Jshell, creating of Java program involves the following steps.
- ◆ Open editor and write program
- ◆ Save the program
- ◆ Compile the program
- ◆ Edit if any compile time error
- ◆ Run the program
- ◆ Edit if any runtime error
- ◆ Repeat the process
- ◆ Jshell does not require above steps. We can evaluate statements, methods and classes, even can write hello program without creating class.
- ◆ JShell helps you try out code and easily explore options as you develop your program

C:\Users\DELL>jshell --help : Help command on JSHELL from your location in cmd
jshell --help from command prompt will let you all the features
C:\Users\DELL>jshell --version : To know the version of Jshell

What is Snippet in Jshell

- ◆ Any valid java statement is called as Snippet in Jshell expect package statement
- ◆ It can be a variable or method or class or interface
- ◆ **/list** -> Will provide all active snippets
- ◆ **/list -start** -> will provide the start up snippets
- ◆ **/list -all** -> will give you all snippets (pre defined and user defined, active and non active). A Snippet becomes non active when you use the same variable name and modify the value then Jshell considers the latest value as the value for that variable and old variable is inactive
- ◆ Ex: `int x=10` -> Currently this is active snippet
- ◆ **Inactive Variable**
- ◆ When I write one more variable with same name like below
- ◆ `String x="Abc";` then when I enter the `/list` command system Jshell will show the value for x as "Abc" and previous `x=10` becomes inactive so this inactive variable can be listed when you use the command `/list -all`
- ◆ Each snippet has snippet id
- ◆ If I enter some value like below `op` is 30

Example:

- ◆ `->jshell>10+20`
- ◆ `Op: $1 =>30`
- ◆ **Scratch Variable:** To hold some temporary values Jshell uses some variables those are called scratch variables
- ◆ Variables created by Jshell is called as Scratch variable
- ◆ On top of this scratch variables you can do the operations on this scratch variables
- ◆ `Jshell>$1>5`
- ◆ `Op is : $2=>>true` (because \$1 is nothing but scratch variable and it is holding the value of 30 and after comparing this `$1>5` again it creates new variable that is \$2 which is true)
- ◆ You can also print the scratch variable
- ◆ `Jshell> System.out.println($1) => output is 30`
- ◆ `Jshell> System.out.println($2)=> output is true`

If you want to list the snippet based on Id

- ◆ `Jshell>/list 1 => list only 1 snippet`
- ◆ `Jshell>/list 1 5 => list only 1 and 5 snippet`
- ◆ `Jshell> /list x => list out x variable`
- ◆ `Jshell> /list add => if add is variable it displays that if it is method then it displays method`

- ◆ **Package declarations is not allowed in Jshell**

To Drop the Snippet (How to Drop a Snippet)

- ◆ `Jshell> /drop snippetid/name`
- ◆ `Jshell> /drop x or /`

To execute the snippet again without typing use below command

- ◆ `Jshell> /snippet id`

What is Explicit Variables: Variables which are created by the programmer

- ◆ `Jshell> int x=10;`
- ◆ `Jshell> String y="karthik"`

What is Implicit Variables or Scratch Variables

- ◆ These variables are the one which are created by the JSHELL (To store the temporary values)
- ◆ Jshell> 10+20
- ◆ \$3=>30 (Here \$3 is the scratch variable)

In JSHELL Two variables with same name is not allowed. If you declare the variable will hold the latest value

- ◆ JSHELL-> overrides the previous value if you try to create the variable with same name:

Example:

- ◆ Jshell> int x=10;
Jshell>System.out.println(x)
- ◆ Op: 10
- ◆ Jshell> String x="ABC"
- ◆ Jshell>System.out.println(x)
- ◆ Op : ABC
- ◆ Here x=10 is int variable it is replaced by String x and value is ABC

To see all the variables in jshell

- ◆ Jshell>/var or /vars will provide all the available variables
- ◆ Jshell> /vars -all => List out all variables (Active and inactive)

To drop particular variable use /drop variable name

- ◆ Jshell> List<String> l== List.of("saas","kasd","sdds");
- ◆ Jshell> /drop l; or /drop snippet id

What is the difference between System.out.println and System.out.printf in JSHELL

- ◆ Jshell>System.out.println("Hello")
- ◆ Op: is Hello (Here no Scratch variable will be created because return type of println method is void)
- ◆ Jshell>System.out.printf("Hello")
- ◆ Op is:
- ◆ Hello\$21 ==> java.io.PrintStream@eec5a4a
- ◆ | created scratch variable \$21 : PrintStream
- ◆ Here printf method return type is not void hence it is creating scratch variable

Working with Methods in Jshell:

Writing method in Jshell:

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

Calling the method using Jshell:

```
jshell> main()  
Hello
```

/List -all

```

22 : public void main()
    {
        System.out.println("Hello");
    }
e3 : main
23 : main()

```

E3: is nothing but error while calling the method I didn't pass the bracehence error

Method overload is possible in JSHELL

To list all methods /methods

```

jshell> public void main(int x)
...> {
...> System.out.println("Ac");
...> }
| created method main(int)

jshell> ./methods
| void add(int,int)
| void main()
| void main(int)

```

- ◆ Look at here .methods is giving main() and main(int x) -> that means method overload is possible
- ◆ If .you write the new method with same signature then old copy will be overridden with new code

```

jshell> int main(int c)
...> {
...> return 0;
...> }
| created method main(int)

```

```

jshell> ./methods
| void add(int,int)
| void main()
| int main(int)

```

- ◆ Here void main(int x) is overridden with int main(int x)
- ◆ Any method you can declare in JSHELL

```

jshell> public void m4()
...> {
...> System.out.prontln(t);
...> }
| created method m4(), however, it cannot be invoked until variable t is declared

```

The above shows I created method and used variable but I haven't declared variable so JSHELL doesn't throw compilation error. That means we can use Undeclared variable in JSHELL

But when u want to call this method m4 until you declare the un declared variable

```

jshell> m4()
| attempted to call method m4() which cannot be invoked until variable t is declared

```

To drop method
/drop method name:

```
jshell> /methods
| void add(int,int)
| void main()
| int main(int)
| void m4()
|     which cannot be invoked until variable t is declared

jshell> /drop m4
| dropped method m4()

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

If multiple method with same name is present then which method will be dropped ?

Answer:

JSHELL will provide the error:

```
jshell> /drop main
| The argument references more than one import, variable, method, or class.
| Use one of:
| /drop 22 : public void main()
| {
|     System.out.println("Hello");
| },
| /drop 25 : int main(int c)
| {
|     return 0;
| }
```

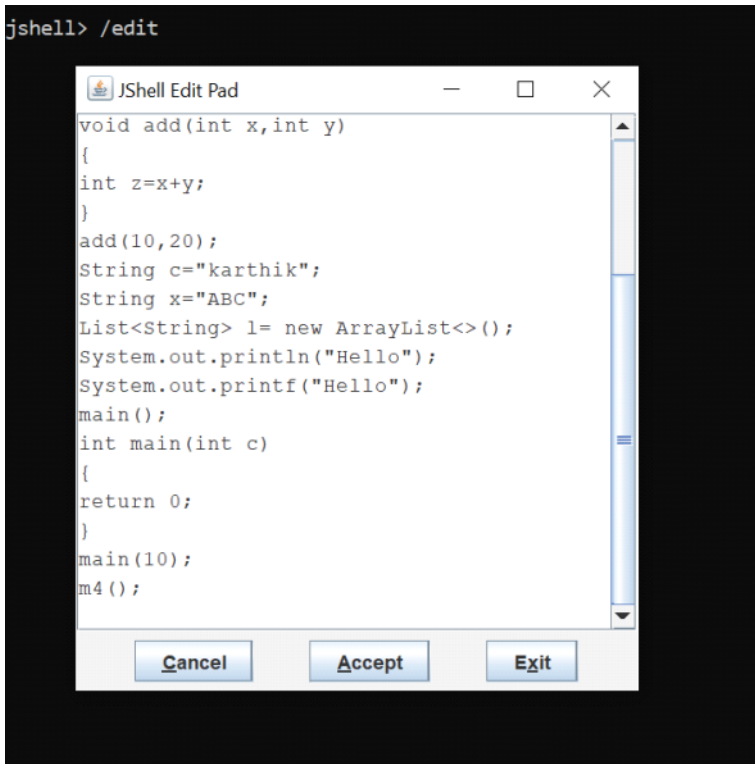
To drop the method in this case you can use /drop snippet id in case when u have multiple methods with same name

```
jshell> /drop 22
| dropped method main()

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

We can Use the JSHELL in built editor to write method

Jshell>/edit => this command will open the jshell in built command



In this editor you can write the methods or any java code and click on accept and click on exit. So that JSHELL is closed.

Then this method or java code is available for JSHELL

JSHELL>

```
jshell> m1()
from JSHELL built in editor
```

See here I have written the method in JSHELL editor and executed from JSHELL

If you want to modify the existing method using JSHELL it is easy if you open the JSHELL editor and modify and then click on Accept. Immediately it gets reflected in JSHELL

To change the editor to notepad or notepad++ then follow the below command
Jshell> /set editor %Editor installed path. If you want notepad then get notepad installed path and give it here%

We can create interface, classes also in JSHELL

If you want to know all classes and interfaces available in JSHELL

JSHELL> /types

```
jshell> interface test{
...> }
| created interface test

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

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

To open the existing JSHELL file.

First you need to create snippets and save as name.jsh (jsh is not mandatory it is recommended)

Jshell> /open test.jsh

To see all commands whether it is loaded from file or not you can use /list command

```
jshell> /open Test1.jsh
jshell> /list
1 : int x=10;
2 : int j=1032;
```

What ever the commands you have entered in JSHELL to store those commands to a file you have to use the below command

JSHELL> save test1.jsh

➤ Interface changes in Java 9: Private methods in Java 9

Java 9 allow us to define private methods in interface. It means now encapsulation is possible in interface also. But we should know what the need of private methods in interface is and when we should use it. Let's take an example:

- ◆ Constant variables
- ◆ Abstract methods
- ◆ Default methods
- ◆ Static methods
- ◆ Private methods
- ◆ Private Static methods
- ◆ Advantage of having the private methods in Java 9 is code reusability with in the interface.

➤ Rules for using the private method in interface

- ◆ We can't make the Private method as an **abstract method**. It means we can't use abstract keyword with a private method.
- ◆ The private method can be a **static method** as well. It means a private method can be static and non-static.
- ◆ We should use private modifier to define these methods and no lesser accessibility than private modifiers.
- ◆ Private method can be used or accessed only inside interface.
- ◆ Private static method can be used inside other static and non-static interface methods.
- ◆ Private non-static methods cannot be used inside private static methods

➤ Try With Resources:

- ◆ This try with resource is introduce in Java 1.7
- ◆ Generally we use try, catch and finally to do the exception handling we use finally block is used to write the clean up code and finally block gets executed every time irrespective of exception or not
- ◆ Before 1.7 we should close the resources using finally block, it is the programmer responsibility and if the programmer forgets then it creates the blocking operations. To overcome try with resource came into picture

What is try with resource ?

- ◆ In 1.7 this Tr with resource is introduced. Execute the below code even though the finally block is not present the JVM will execute the finally block or closing the connection code etc.

Advantages of Try With Resource is

- ◆ Better Readable code

- ◆ No need to add null checks

Rules for Try with resource in Java 1.7 is

- ◆ In Java SE 7 or 8 versions, we should follow these rules to use Try-With-Resources statement for Automatic Resource Management(ARM)
- ◆ **Any Resource (which Pre-defined Java API class or User Defined class) must implement java.lang.AutoCloseable.**
- ◆ Resource object must refer either final or effectively final variable
- ◆ **If Resource is already declared outside the Try-With-Resources Statement, we should re-refer with local variable (As shown in the below example)**
- ◆ That newly created local variable is valid to use within Try-With-Resources Statement.

Issue with 1.7 Try with resource is

- ◆ We cannot use any Resource (which is declared outside the Try-With-Resources) within try() block of Try-With-Resources statement.

Example:

```

10
11 void testARM_Before_Java9() throws IOException{
12     BufferedReader reader1 = new BufferedReader(new FileReader(
13         try (reader1) {
14             System.out.println(reader2.readLine());
15         } catch (Exception e) {
16             System.out.println("exception");
17         }
18     }
19

```

Enhance made in Java 9 for try with resource is that try() can take multiple resource and there is no need to create the resource inside try(); we can declare the resource anywhere and we can use that resource in try();

In Java SE 9, if we have a resource which is already declared outside the Try-With-Resource Statement as final or effectively final, then we do NOT need to declare a local variable. We can use previously created variable within Try-With-Resource

```

12 void testARM_Before_Java9() throws IOException{
13     BufferedReader reader1 = new BufferedReader(new FileReader("C:\\Users\\DELL\\Desktop\\Ja
14     BufferedReader reader2 = new BufferedReader(new FileReader("C:\\Users\\DELL\\Desktop\\Ja
15     try (reader1;reader2) {
16         System.out.println(reader1.readLine());
17         System.out.println(reader2.readLine());
18     } catch (Exception e) {
19         System.out.println("exception");
20     }
21

```

Syntax: try(R){ --//Here R is nothing but resource. Generally we don't have try with brackets. It is introduced in java 1.7
 }catch(Exception e){
 }

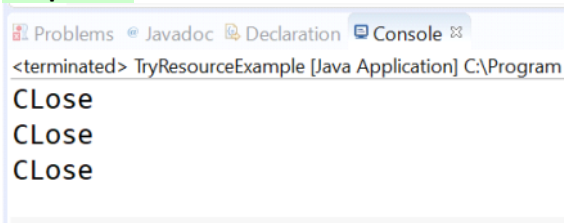
The try() automatically calls the close method (close method is present in Autoclosable interface.)so we can create the user defined class and implement the autoclosable interface and override the close method. Try multiple resources and use the try with resource automatically close method will be called for all the resources.

```

7 public class TryResourceExample implements AutoCloseable{
8     public static void main(String[] args) throws FileNotFoundException {
9         TryResourceExample t= new TryResourceExample();
10        TryResourceExample t1= new TryResourceExample();
11        TryResourceExample t2= new TryResourceExample();
12        try(t;t1;t2){
13
14        }catch (Exception e) {
15            System.out.println("Exception");
16        }
17    }
18    @Override
19    public void close() throws Exception {
20        System.out.println("Cclose");
21    }
22 }

```

Output is :



```

<terminated> TryResourceExample [Java Application] C:\Program
Cclose
Cclose
Cclose

```

➤ Process API Improvements

Java 9 has made some changes to the Process API, through which working with operating system becomes easy.

For Example: If we would like to know the details of specific process or kill the process or current running process or create new process then this API is useful.

- ◆ Java 9 Process API Updates: –
- ◆ Some new methods are added to the Process class present in java.lang package.
- ◆ The process class is an old class in Process API, only some methods like, pid(), info() etc are added in java 9.
- ◆ Methods like startPipeline() are added to ProcessBuilder class which is already present in old java versions.
- ◆ ProcessHandle(l) : This is a new interface added in java 9. It is used to handle a process.
- ◆ ProcessHandle.Info(l): It is an inner interface, it provides all the information related to a Process
- ◆ To get the information on current running process we have to call the static method called current() present in the ProcessHandle Interface.

```
public class ProcessApiEnhancements {
```

```
    public static void main(String[] args) {
```

```
        ProcessHandle pr=ProcessHandle.current();
        System.out.println(pr.pid());
        System.out.println(pr.info().user());
        System.out.println(pr.info().command().get());
```

- ◆ If you execute the above code, you can get the details of current running process.

```
        ProcessHandle pr1=ProcessHandle.of(30632).get();
        System.out.println(pr1.pid());
        System.out.println(pr1.info().user());
```

- ◆

```
        System.out.println(pr1.info().command().get());
        System.out.println(pr1.info().startInstant().get());
        System.out.println(pr1.info().totalCpuDuration().get());
        pr1.destroy();
```

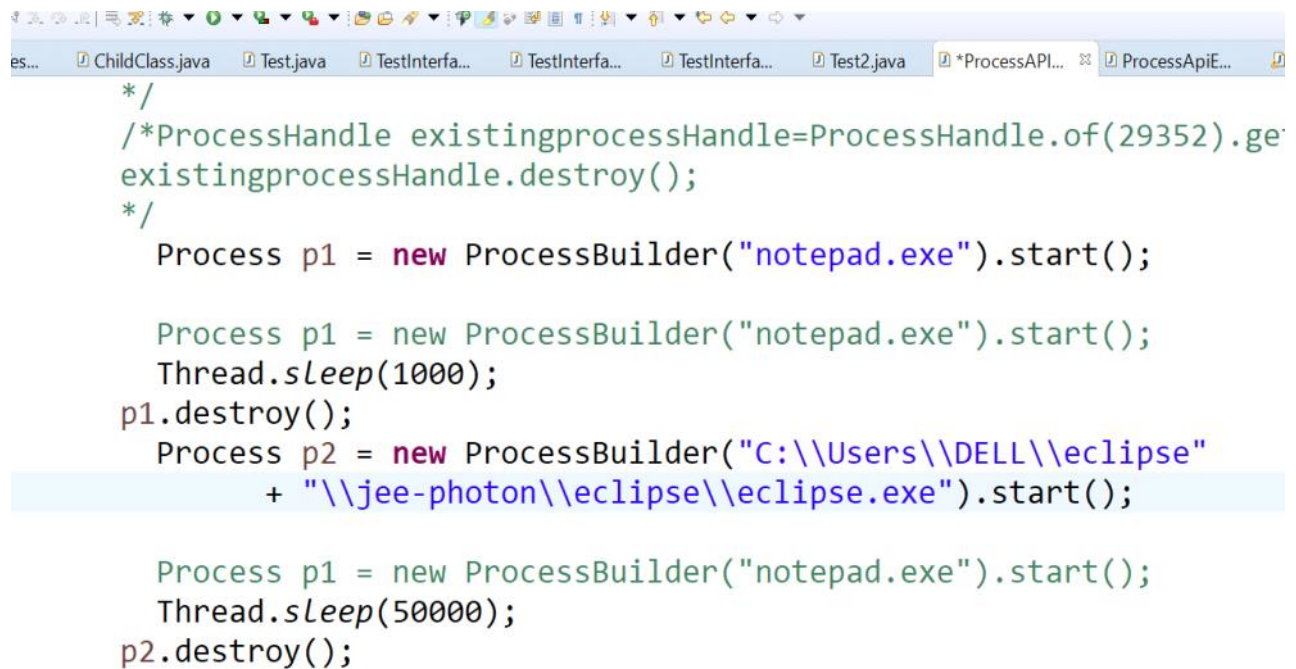
- ◆ If we want to know the details of some process based on process id then we can use the of static method of ProcessHandle interface.
- ◆ If we want to destroy the process which is same as clicking the end process button on Task manager then we have to call the destroy method of ProcessHandle.
- ◆ If we would like to know all the process information then below code will work. We have to call the allProcesses method. This is same as opening the task manager and looking at details tab.

```
        System.out.println("All Process information");
        Stream<ProcessHandle> processStream=ProcessHandle.allProcesses();
        processStream.forEach(p->{
```

- ◆

```
            System.out.println(p.pid());
            System.out.println(p.info().user());
            System.out.println(p.info().command().isPresent());
        });
```

- ◆ To create new process, like opening a notepad or eclipse then we have to go to start command and search for the process and start it. But using the java program u can open the process. Look at the below example.
- ◆ To start the process we should use the class called as ProcessBuilder and provide the .exe path of the application which you want to start.
- ◆ When you execute the below code it starts the notepad process and eclipse process.



```

    */
    /*ProcessHandle existingprocessHandle=ProcessHandle.of(29352).get
    existingprocessHandle.destroy();
    */
    Process p1 = new ProcessBuilder("notepad.exe").start();

    Process p1 = new ProcessBuilder("notepad.exe").start();
    Thread.sleep(1000);
    p1.destroy();
    Process p2 = new ProcessBuilder("C:\\Users\\DELL\\eclipse"
        + "\\jee-photon\\eclipse\\eclipse.exe").start();

    Process p1 = new ProcessBuilder("notepad.exe").start();
    Thread.sleep(50000);
    p2.destroy();

```

➤ Java Platform Module System

- ◆ What is Module ? Module is nothing but group of packages

A Module is a group of closely related packages and resources along with a new module descriptor file.

- ◆ With new Java 9 modules, we will have better capability to write well-structured applications. This enhancement is divided into two areas:
- ◆ Modularize the JDK itself.
- ◆ Offer a module system for other applications to use
- ◆ Java 9 Module System has a "java.base" module. It's known as Base Module. It's an Independent module and does NOT dependent on any other modules. By default, all other modules are dependent on "java.base".
- ◆ In Java 9, modules help us in encapsulating packages and managing dependencies. So typically,
- ◆ a class is a container of fields and methods
- ◆ a package is a container of classes and interfaces
- ◆ a module is a container of packages
- ◆ From Java 9 onwards, public means public only to all other packages inside that module. Only when the package containing the public type is exported, can it be used by other modules
- ◆ Package statement is mandatory in module based program
- ◆ The main goal of the Java 9 module system is to support modular programming in Java.
- ◆ Currently, the Java 9 module system has about 98 modules, but it is continuing to evolve. Oracle has categorized JDK jars and Java SE specifications into two sets of modules.
- ◆ The default module for all JDK and user-defined modules is the base module java.base. It is an independent module and doesn't depend on any other modules. java.base is known as the "mother of Java 9 modules."

Module Descriptor

When we create a module, we include a descriptor file that defines several aspects of our new module:

- ◆ Each module has a unique name
- ◆ Each module has some description in a source file
- ◆ The module descriptor file is placed in the top-level directory
- ◆ Each module can have any number of packages and types
- ◆ One module can depend on any number of modules
 - **Name** – the name of our module
 - **Dependencies** – a list of other modules that this module depends on
 - **Public Packages** – a list of all packages we want accessible from outside the module
 - **Services Offered** – we can provide service implementations that can be consumed by other modules
 - **Services Consumed** – allows the current module to be a consumer of a service
 - **Reflection Permissions** – explicitly allows other classes to use reflection to access the private members of a package

The module naming rules are similar to how we name packages (dots are allowed, dashes are not).

We need to list all packages we want to be public because by default all packages are module private.

Module Types

There are four types of modules in the new module system:

- **System Modules** – These are the modules listed when we run the *list-modules* command above. They include the Java SE and JDK modules.
- **Application Modules** – These modules are what we usually want to build when we decide to use Modules. They are named and defined in the compiled *module-info.class* file included in the assembled JAR.
- **Automatic Modules** – We can include unofficial modules by adding existing JAR files to the module path. The name of the module will be derived from the name of the JAR. Automatic modules will have full read access to every other module loaded by the path.
- **Unnamed Module** – When a class or JAR is loaded onto the classpath, but not the module path, it's automatically added to the unnamed module. It's a catch-all module to maintain backward compatibility with previously-written Java code.

To set up a module, we need to put a special file at the root of our packages named *module-info.java*.

This file is known as the module descriptor and contains all of the data needed to build and use our new module.

Advantages of Module over jar:

Reliable configuration—Modularity provides mechanisms for explicitly declaring dependencies between modules in a manner that's recognized both at compile time and execution time. The system can walk through these dependencies to determine the subset of all modules required to support your app.

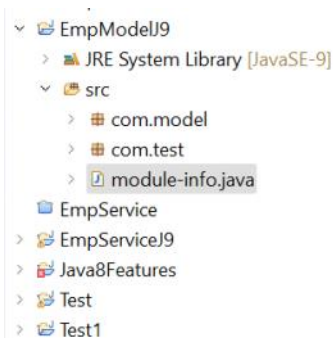
Example: Whenever you are working with multiple projects (Project A and Project B) and if one project (Project B) would like to access the file from another project (Project A) then we have to place the jar file of Project A in project B class path, then we can access all the files of Project A.

Strong Encapsulation and Security: Other modules can access only exported packages and not all packages.

Scalable Java platform: Compared to Jar module occupies less space as all the class will not be included in jar only exported packages will be part of the Jar

➤ An exports clause in Module-info.java

- **Export can only be used for packages**
- A module can export its packages to the outside world so that other modules can use them. In the module descriptor, you use the exports clause to export packages to the outside world or to other modules:
- In the below image, you can see that EmpModelJ9 project has 2 packages but out of those 2 packages we are exporting only one package so that this package com.model can be accessed by any other module.
- Since we are not exporting com.test so other module can make use if the classes present in this package.
- That means in Java 9 even though I mention the class as Public, it means that until we mention that package as exports in module-info.java other module cannot be able to access. Public class access is limited to for that specific module if we don't specify export in module-info.java
- In Java 9 Module, there is a mechanism to control the access to packages whereas this kind of option is not present in jar mechanism



```
5 * @author DELL
6 *
7 */
8 module EmpModelJ9 {
9     exports com.model;
10 }
```

➤ Qualified Export:

- Whenever we use Export keyword followed by Package name that package will be available to all other modules. But if my requirement is to make this package available for specific module then such type of dependency is called as Qualified export.

Example:

- Syntax: Export package Name to Module Name.
- Export Pack1 to ModuleB
- Here we can even mention multiple modules also
- Export Pack1 to ModuleB, Module C

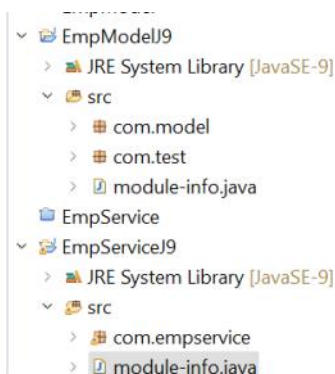
module-info.java:

```
1) module exportermodule
2) {
3)     exports pack1;
4)     exports pack2 to moduleA;
5)     exports pack3 to moduleA,moduleB;
6) }
```

| | moduleA | moduleB |
|-------|---------|---------|
| pack1 | ✓ | ✓ |
| pack2 | ✓ | ✗ |
| pack3 | ✓ | ✓ |

➤ A requires clause

- A module can import or use other modules packages. In the module descriptor, you use the requires clause to import other modules in order to use their packages:
- Requires directive can only be used for Modules
- Requires keyword followed by one module only . If we want to include multiple modules then for each module we should specify requires keyword
- Requires ModuleA;
- Requires ModuleB;
- **We cannot mention moduleA, Module B -> This is wrong**
- If one module requires another module then you need to mention that using requires attribute
- In the below example, EmpServiceJ9 module is dependent on EmpCommonJ9 module hence in EmpServiceJ9 project module-info.java file I have to mention that requires EmpCommonJ9
- **requires <ModuleName>**



```
5 * @author DELL
6 *
7 */
8 module EmpServiceJ9 {
9     requires EmpCommonJ9;
10 }
```

➤ Requires Static Module Name:

- This dependency can be used only when at we need the specified module to be available at compile time and not on run time that means at run time module name is optional.
- Static key word is used to provide optional dependencies. Anything specified with requires static then that corresponding module is available at compile time but at run time is not needed
- Cyclic Dependencies:
- B extends A and A extends B -> This is called cyclic inheritance and it is not allowed in Java.
- Similarly -> Module A requires Module B and Module B requires Module A -> This is cyclic dependency. This is not allowed Java Modules.
- EmpModelJ9 and EMPBusinessServiceJ9 are 2 java 9 modules. If I am mentioning the cyclic dependency then below error that compiler shows.
- Module dependency can occur among more than 2 modules that means Module A-> Module B
- Module B-> Module C
- Module C- Module A


```

8 module EmpModelJ9 {
9     exports com.model;
10    requires EmpBusinessServiceJ9;
11 }

```

⚠ Cycle exists in module dependencies, Module EmpModelJ9 requires itself via EmpBusinessServiceJ9
Press 'F2' for focus

Module Graph is nothing but dependency between the module is represented using module graph.

➤ Uses

- Using uses keyword we can specify that our module needs or consumes some service. Service is a interface or abstract class. It should not be an implementation class.

• Syntax

- uses <service-required>;

Example

```

module com.module.util{
    uses com.util.PersonDataService;
}

```

- **Note:** The most important thing to note here is that 'requires' adds a module dependency, whereas 'uses' specifies required service class.

➤ Provides ... With

- We can specify that our module provides some services that other modules can use.

Syntax

- provides <service-provided> with <service-implementation-class> ;

Example

- module com.module.util{
 provides com.util.PersonDataService with
 com.util.DbPersonServiceImpl;
}

➤ Open

- Since java 9 encapsulation and security is improved for the reflection apis. Using reflection we were able to access even the private members of the objects.
- From java 9 this is not open by default. We can although grant reflection permission to other modules explicitly.
- open module com.module.util{
}
- In this case all the packages from util module are accessible using reflection.

➤ Opens

- If we do not want to open all the packages for reflection we can

specify packages manually using 'opens' keyword.

- ```
module com.module.util{
 opens com.module.package1;
}
```
- In this case only classes from package1 are accessible using reflection.

#### ➤ Opens ... To

- Using 'opens ...to' keyword we can open reflection permission for specific packages to specific modules only.
- ```
module com.module.util{
    opens com.module.package1 to module.a, module.b,
    org.test.integration;
}
```
- In this case only module.a, module.b, org.test.integration modules can access classes from package1 using reflection.
- **Note:** If we need reflection access to the module, we can gain the access using command line option '--add-opens', even if we are not the owner of the module

➤ Aggregator Module

- This is nothing but a module which is going to club multiple modules and make it available for next module is called as aggregator module.
- The transitive keyword is important in this aggregator module. This provides the reusability
- First of all, this is not a technical concept. It is just a convenience concept for developers to make their life easier.
- Sometimes multiple modules require other multiple modules. Instead of adding these in every module descriptor, we can create one module that will add all the required dependency using 'transitive'. Then we just need to add dependency of this module wherever needed, this will add all the required modules transitive dependency. This common module is the 'Aggregator module'.
- For example, we have 10 modules, modA to modJ. modP, modQ, modR needs all 10 modules, then we can create one common module as below,

```
module modulePQR{
    requires transitive modA;
    ....
    ...
    requires transitive modJ;
}
```

Then modules P, Q and R just need to add require for modulePQR

```
module modP{
    requires transitive modulePQR;
}
```

The module modulePQR is the Aggregator module.

➤ Package Naming conflicts

- 2 Jar files can contain the same package name which cause the problem of version control and abnormal execution of the program where as
- 2 modules cannot contain the same package name, if it is there compiler will throw compilation error. Assume that Module A, Module B and Module C are 3 modules and Module A and B has same package that is pack1. Execute the below example

```
Module ModuleA{
```

```
Exports pack1;
```

```
}
```

```
Module ModuleB{
```

```
Exports pack1;
```

```
}
```

```
Module C{
```

```
Requires ModuleA;
```

```
Requires ModuleB;
```

```
}
```

- JVM is going to evaluate whether all the dependent modules are available or not at beginning only
- **Module Resolution Process: In case of traditional class path, JVM wont check for required .class files at the beginning. While execution of the program if JVM requires any .class file then JVM will search in classpath and if it is available then it will load**
- **But in Module programming, JVM will search for 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 the execution. That's the reason there is no chance of getting noClassDef found error in the middle of program execution**

➤ Collection API Enhancements:

- ◆ If we want to create immutable collection using java 8 then we have to follow the below steps.

```
8 public class ImmutableCollectionJava8 {
9     public static void main(String[] args) {
10         List<Integer> l= new ArrayList<Integer>();
11         l.add(10);
12         l.add(20);
13         l.add(30);
14         l=Collections.unmodifiableList(l);
15     }
16 }
```

- ◆ Here UnModifiableList or UnModifiableCollection will provide the Immutable Collection object
- ◆ After line Collections.UnModifiableList() if we perform the l.add() or l.remove() then below exception is thrown.

```

3=import java.util.ArrayList;
4 import java.util.Collections;
5 import java.util.List;
6 public class ImmutableCollectionJava8 {
7=     public static void main(String[] args) {
8         List<Integer> l= new ArrayList<Integer>();
9         l.add(10);
10        l.add(20);
11        l.add(30);
12        l=Collections.unmodifiableList(l);
13        l.add(40);
14    }
15 }
16

```

Exception in thread "main" [java.lang.UnsupportedOperationException](#)
 at java.util.Collections\$UnmodifiableCollection.add(Unknown Source)
 at com.practice.ImmutableCollectionJava8.main([ImmutableCollectionJava8.java:13](#))

- ◆ In Java 9, you can create immutable collections such as immutable list, immutable set and immutable map using new factory methods, no need to write the code like above
- ◆ The List.of() static factory methods provide a convenient way to create immutable lists. The List instances created by these methods have the following characteristics:
- ◆ This Of() method is overloaded method and it is present in List interface.
- ◆ They are structurally immutable. Elements cannot be added, removed, or replaced. Calling any mutator method will always cause UnsupportedOperationException to be thrown
- ◆ They disallow null elements. Attempts to create them with null elements result in NullPointerException.
- ◆ They are serializable if all elements are serializable.
- ◆ The order of elements in the list is the same as the order of the provided arguments, or of the elements in the provided array.

```
List<String> l= List.of("Jan", "Feb", "Mar", "Apr", "May", "Jun", "July", "Aug", "Sep", "Oct", "Nov", "Dec");
System.out.println(l);
```

```
Set<String> l1= Set.of("Jan", "Feb", "Mar", "Apr", "May", "Jun", "July", "Aug", "Sep", "Oct", "Nov", "Dec");
System.out.println(l1);
```

➤ Diamond Operator for Anonymous Inner Class

- ◆ Before 1.5 If we want to create the collection object, we may create like below
 - ◆ List l= new ArrayList();
- ◆ Problem with this statement is that we can add any type of objects to this list. Example: we can add String, Integer and Employee. When we add the different objects like this the problem is that when you want to retrieve the value we need to explicitly type cast the value. Apart from the type cast, we also don't know what kind of object is present at which position that is type safety.
- ◆ To resolve the type safety and type cast problem in 1.5 generics came into

picture so the above statement became like below.

```
List<Integer> l= new ArrayList<Integer>();
```

- ◆ In the above statement <> is called as diamond operator so here when we create the array list object we are specifying the Object type, so that we can add only Integer objects and we cannot add any other objects. In this way type safety is achieved. By mentioning like this when you have to get the value we don't need to cast it to specific type.
- ◆ We can use like this Integer i=l.get(3), but the point is here we need to specify the diamond operator with generic type for both the sides
- ◆ Change happened in 1.7 is that we don't need to specify the generic type towards right side.

In 1.9 the change is introduced is we can use the diamond operator for anonymous inner classes. See the code below

- ```
public abstract class DiamondAnstract<T> {
 abstract T add(T num1, T num2);
}

public class DiamondImpl {
 public static void main(String[] args) {
 DiamondAnstract<Integer> d= new DiamondAnstract<Integer>() {
 @Override
 Integer add(Integer num1, Integer num2) {
 return Integer.valueOf(num1.intValue()+num2.intValue());
 }
 };
 System.out.println(d.add(10, 12));
 }
}
```
- ◆ Here in the above code we were able to provide the diamond operator with generics for anonymous inner class.

## ➤ Stream API Enhancements

- ◆ JDK 9 adds four methods to the Stream interface.

| Method Name  | Method Type | Return Type | Arguments                                                         | Description                                                                                                                                                                                                                                                                                                     |
|--------------|-------------|-------------|-------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| takeWhile()  | Default     | Stream<T>   | Predicate<? super T>                                              | <p>If calling stream is ordered, then this method returns a stream containing first <i>n</i> elements of the calling stream which satisfy the given predicate.</p> <p>If the calling stream is unordered then this method returns all or some elements which satisfy the given predicate.</p>                   |
| dropWhile()  | Default     | Stream<T>   | Predicate<? super T>                                              | <p>If the calling stream is ordered, then this method drops first <i>n</i> elements which satisfy the given predicate and returns remaining elements.</p> <p>If the calling stream is unordered, then this method returns remaining elements after dropping the elements which satisfy the given predicate.</p> |
| ofNullable() | Static      | Stream<T>   | T t                                                               | <p>This method takes one element as an argument and returns a Stream containing that single element if the passed element is non-null.</p> <p>If the passed element is null, it returns an empty Stream.</p>                                                                                                    |
| iterate()    | Static      | Stream<T>   | T seed,<br>Predicate<? super T> hasNext,<br>UnaryOperator<T> next | It returns a Stream produced by next starting from seed till hasNext returns true.                                                                                                                                                                                                                              |

- First, there are two related methods: **takeWhile(Predicate)** and **dropWhile(Predicate)**. These methods are complementary to the existing **limit()** and **skip()** methods, but they use a **Predicate** rather than a fixed integer value

#### takeWhile(Predicate)

- The takeWhile() method continues to take elements from the input stream and pass them to the output stream until the test() method of the Predicate returns true. It is also a short-circuiting intermediate operation.
- Takes values while the filter is true, then stops
- takeWhile is similar to Limit() method.

#### Difference between filter and takeWhile():

- filter will remove all items from the stream that do not satisfy the condition.

#### Example:

```
Stream.of(1,2,3,4,5,6,7,8,9,10,9,8,7,6,5,4,3,2,1)
 .filter(i -> i < 4)
 .forEach(System.out::print);
```

```
System.out.println("\n filter OP");
```

- Stream.of(1,2,3,4,5,6,7,8,9,10).filter(i -> i % 2 == 0 )  
 .forEach(System.out::print);

- The Output will be all the element which satisfy the condition. **Output for 1st program is : 123321**
- The Output for 1st program is : 246810**
- takeWhile will abort the stream on the first occurrence of an item which does not satisfy the condition.
- Takewhile is similar to limit method. Limit method takes long as argument. So what ever the long variable that is mentioned in limit method, stream will limit the stream up to that argument
- Similarly takeWhile() method will take predicate interface. Once the

predicate condition false, it won't check the remaining elements.

Example:

```
IntStream.of(1, 10, 100, 1000, 10000, 1000, 100, 10, 1, 0, 10000)
 .takeWhile(i -> i < 5000)
 .forEach(System.out::println);
```

- ♦ Output of this statement is 1,10,100,1000
- ♦ When 10000 is occurred, 10000<5000 is false so it will stop the logic and the out put is 1,10,100,1000
- ♦ In the same example if I use filter then it will iterate all over the stream and displays the output

```
IntStream.of(1, 10, 100, 1000, 10000, 1000, 100, 10, 1, 0, 10000)
 .filter(i -> i < 5000)
 .forEach(System.out::println);
```

- ♦ **The difference between filter and take while is, when we use filter it evaluate the condition over entire stream. In Takewhile when the condition fails then will not check for remaining elements. As the name suggests the stream will iterate while the condition is true**
- ♦ Other takeWhile Examples are :

```
System.out.println("\n Limit OP");
Stream.of(1,2,3,4,5,6,7,8,9,10,9,8,7,6,5,4,3,2,1).limit(4)
 .forEach(p->System.out.println(p));
System.out.println("\n take while");
```

♦

```
Stream.of(1,2,3,4,5,6,7,8,9,10,9,8,7,6,5,4,3,2,1)
 .takeWhile(i -> i < 4)
 .forEach(System.out::print);
```

```
System.out.println("\n take while OP");
```

- ♦ 

```
Stream.of(1,2,3,4,5,6,7,8,9,10).takeWhile(i -> i % 2 == 0)
 .forEach(System.out::println);
```

### dropWhile(Predicate)

- ♦ The dropWhile() method does the opposite; it drops elements from the input stream until the test() method of the Predicate returns true. All remaining elements of the input stream are then passed to the output stream. It is also a short-circuiting intermediate operation.
- ♦ It is same as skip method. Skip method takes long parameter. Till this parameter the elements are skipped. Similarly dropWhile method skips the elements until the condition is satisfied.
- ♦ We want to drop all elements till we encounter an element that satisfies the predicate. From that element onwards, all the remaining elements in the stream should be processed.
- ♦ Knowing the difference between an ordered and unordered stream is important before getting into the takeWhile and dropWhile methods as they operate differently for them.
- ♦ An ordered stream is one that is created out of a collection that produces a stream with encounter order – example, List and array. Since a list or an array is ordered, the stream created out of them is ordered.

- ◆ For other collection sources like a HashSet or a HashMap, which doesn't have a guaranteed order, the stream created with them as the source results in an unordered stream.

- ◆ 

```
list.stream()
 .dropWhile(element -> element.equals("f"))
 .forEach(System.out::println);
```
- ◆ 

```
IntStream.of(1, 10, 100, 1000, 10000, 1000, 100, 10, 1, 0, 10000)
 .dropWhile(i -> i < 5000)
 .forEach(System.out::println);
```

### ofNullable()

- ◆ ofNullable() is a static method which takes one element as an argument and returns a Stream containing that single element if the passed element is non-null. If the passed element is null, it returns an empty Stream

- ◆ 

```
long count = Stream.ofNullable(25).count(); //Non-null element
System.out.println(count); //Output : 1
```
- ◆ 

```
count = Stream.ofNullable(null).count(); //Null element
System.out.println(count); //Output : 0
```

### iterate()

- ◆ iterate() method is already there in Stream interface from Java 8. But, Java 9 provides another version of iterate() method which takes an extra argument hasNext of type Predicate which decides when to terminate the operation.
- ◆ iterate() method is also a static method.

```
Stream.iterate(1, i -> i <= 100000, i -> i*10).forEach(System.out::println);
```

The above condition is same as below without streams

```
for(int i=1;i<100000;i=i*10) {
 System.out.println(i);
}
```

- ◆ Here the above iterate method is taking 3 arguments, till the second condition is satisfied the stream will iterate and does the work mentioned in 3rd parameter.



| Modifier and Type       | Method                                                                 | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|-------------------------|------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| static <T><br>Stream<T> | iterate(initial_element;<br>predicate_expression;<br>update_statement) | <p>Returns a sequential ordered Stream produced by iterative application of the given update_statement to an initial element, conditioned on satisfying the given predicate_expression.</p> <ol style="list-style-type: none"> <li>1. initial_element - the initial element.</li> <li>2. predicate_expression - a predicate to apply to elements to determine when the stream must terminate.</li> <li>3. update_statement - a function to be applied to the previous element to produce a new element.</li> </ol> |
| static <T><br>Stream<T> | iterate( initial_element,<br>update_statement)                         | <p>Returns an <b>infinite</b> sequential ordered Stream produced by iterative application of an update_statement to an initial_element. <b>Be careful when using this, it generates infinite Stream.</b></p> <ol style="list-style-type: none"> <li>1. initial_element - the initial element</li> <li>2. update_statement - a function to be applied to the previous element to produce a new element.</li> </ol>                                                                                                  |