Chapter 7

Packages

- In this chapter, You will learn
  - Definition and Need of packages in project
  - Creating package programmatically & linking it with class
  - Need of javac tool option "-d"
  - Class loader working functionalities
  - Organizing classes as packages
  - Creating sub packages
  - Using other package members
  - Understanding fully qualified name
  - Understanding Import statement
  - Using sub package members
  - Java source file structure
  - Inbuilt packages
- By the end of this chapter- you will understand creating and using package members from same and different package members with proper rules.

Interview Questions

By the end of this chapter you answer all below interview questions

Package keyword

- Definition and need of package.
- Package rule
- Creating packages manually and programmatically
- When should we call a folder is a package?
- Class path settings to access package members from other packages.
- How can we store multiple public and non-public classes in a single package?
- Defining Sub-packages.
- What is the fully qualified name of the class?

Import keyword

- Using existed packages
  a. Fully qualified name
  b. import statements
- What are the difference between
  a. import <packagename>.*;
  b. import <packagename>.<classname>;
- import statement rule
- What is the information import statement provides?
- Is import statement load class into JVM?
- Give a scenario that force you to use both fully qualified name and import statement to access a class/ interface.
- Accessing packaged classes from non-packaged class?
- Accessing non-packaged classes from packaged class?
- importing sub packages
- static imports
- Java program source file structure
- Inbuilt packages.

- Accessibility modifiers for package members to access from other packages
  a. private
  b. <default>
  c. protected
  d. public
- protected accessibility modifier rule

So far we have learnt creating class individually. In this chapter we will learn how can group classes and how to separate one class from other class if both have same name using package.

Definition and Need of package

A folder that is linked with java class is called package. It is used to group related classes, interfaces and enums. Also it is used to separate new classes from existed classes. So by using package we can create multiple classes with same name, also we can create user defined classes with predefined class names.

Package creation

To create a package we have a keyword called "package".

Syntax

```
package <package name>;
```

For example: package p1;

Rule on package statement

package statement should be the first statement in a java file.

Default package

package statement is optional. If we define a class without package statement, then that class is said to be available in "default package i.e.; current working directory".

Below program shows creating a class with package

//Example.java

```
package p1;
class Example{
        public static void main(String[] args) {
                System.out.println("In Example main");
        }
}
```

Compilation

Compiler does not create package physically in current working directory just with *javac* command. Packaged classes must be compiled with "-d" option to create package physically.

Syntax: *javac –d <path in which package should be copied> source filename*

For Example

> **javac -d . Example.java**

With this command compiler creates package "p1" with "Exmple.class" and places it in current working directory. Operator "." represents current working directory.

> **javac -d C:\test Example.java**

With this command compiler creates package "p1" with "Exmple.java" in C:\test folder.

Rule: test folder must be existed in C drive before this program compilation, else it leads to CE.

"-d" functionality

Its actual functionality is creating package with the name mentioned in java file and moving all .class files in that package, and finally storing that package in the given path.

Packaged class code changed by compiler

After compilation compiler replaces class name and its constructor name with its *packagename.classname*. It is called *fully qualified* name.

Check below diagram

//Example.java DWC

```
package p1;

class Example{
        public static void main(String[] args){
                System.out.println("Hi");
        }
}
```

//Example.class CCC

```
class p1.Example extends java.lang.Object{
        p1.Example(){
                super();
        }
        public static void main(String[] args){
                System.out.println("Hi");
        }
}
```

Execution:

We must use package name in executing a packaged class else it leads to exception

```
>java p1.Example
Hi
```

Q) Why we must use package name in execution?

As you observed, in Example.class the class is changed as p1.Example. Since name is p1.Example, it must be executed with the same name means in execution we must use package name.

Q) Can we execute a class from CWD that is placed in another directory?

No, it leads to exception "java.lang.NoClassDefFoundError"

For example

```
>javac -d C:\test Example.java
>java p1.Example
Exception in thread "main" java.lang.NoClassDefFoundError: p1/Example
```

If package is placed in another directory, we must update its path in Classpath environment variable, else it leads above exception.

Updating Classpath environment variable

We can update Classpath in 3 ways

1. By using java command option "–cp" or "–classpath"
2. By using "Set Classpath" command
3. By using "Environment Variables window"

In *first approach* we will have Classpath setting only for the *current java command* execution
In *second approach* we will have Classpath settings only for the *current command prompt*
In *third approach* we will have Classpath settings for *all command prompts forever*.

Usage:

```
>javac –d C:\test Example.java
>java –cp C:\test p1.Example
Hi
        or

>java –classpath C:\test p1.Example
Hi

>Set Classpath=C:\test
>java p1.Example
Hi
```

ClassLoader working functionality

- ClassLoader loads classes into JVM based on Classpath environment variable setup.
- To load class bytes into JVM it searches in the folders those are configured in Classpath environment variable. It searches all folders till it finds given class's ".class" file. If it not found in any one of the folder then it throws "*NoClassDefFoundError*" exception.
- If Classpath is not at all created, then it loads classes only from current working directory.
- If Classpath is created it is mandatory to place "." operator to load classes from current working directory.
- If the classpath environment variable has the character ";" not as a separator, it is treated as "." so that ClassLoader loads classes from current working directory.

Find out from which folder class is loaded and executed from the below syntax

- set classpath=C:\test <= class is loaded from C:\test
- set classpath=. <= class is loaded from current working directory
- set classpath=.;C:\test <= class is loaded from current working directory
- set classpath=C:\test;. <= class is loaded from C:\test
- set classpath=NareshIT;C:\test <= class is loaded from C:\test
- set classpath= NareshIT;;C:\test <= class is loaded from current working directory
- set classpath=NareshIT;C:\test; <= class is loaded from C:\test
- set classpath=NareshIT <= JVM throws "java.lang.NoClassDefFoundError"
- set classpath=NareshIT; <= class is loaded from current working directory

Q) When we are using a class from another class, should I compile that class first?
No need to compile. Compiler automatically compiles that class. For example assume we are calling Example class method from Sample class method we can compile Sample class directly without compiling Example class.

Compiler follows below procedure to compile Example class

1. First it searches for that Example class definition in Sample.java, if not found
2. It searches for Example.class in Sample class package, if not found
3. It searches for Example.java in Sample class package , if not found
4. It searches for Example.class in imported packages, if not found
5. It searches for Example.java in the imported packages, if not found
6. Then compiler terminates Sample.java file compilation by throwing CE: cannot find symbol
7. If Example.java is found, it searches for Example class definition in Example.java file. If it is found, compiler compile entire java file, it means it also compiles other class definitions and generates those class's .class files. Else terminates Sample.java file compilation with above compilation error.
8. If Example.class is found, it also searches for Example.java. If not found, compiler uses Example.class file directly.
9. If Example.java is also available, it checks modified time of both files, if Example.java file modified date is greater than Example.class modified date, it compiles Example.java again for generating Example.class with its latest changed java code.

Test all above points using below two programs

Case #1: Example class definition in another java file

Example.java

```
class Example
{
        static int a = 50;
        static int b = 60;
}
```

Sample.java

```
class Sample{
        p s v main(String[] args){
                Sopln(Example.a);
                Sopln(Example.b);
        }
}
```

Output

```
>javac Sample.java
>java Sample
50
60
```

Case #2: change "a" and "b" variables to 70 and 80 and save Example.java. Then compile and execute Sample.java file directly. Now Example.class is regenerated with new values.

Example.java

```
class Example
{
        static int a = 70;
        static int b = 80;
}
```

Sample.java

```
class Sample{
        p s v main(String[] args){
                Sopln(Example.a);
                Sopln(Example.b);
        }
}
```

Output

```
>javac Sample.java
>java Sample
70
80
```

Case #3: Define Example class in Sample.java with "a" and "b" values as 15 and 16. Then compile and execute Sample.java file directly. Now Example.class is regenerated with the Example class values defined in Sample.java – local preference.

Example.java

```
class Example
{
        static int a = 70;
        static int b = 80;
}
```

Sample.java

```
class Sample{
        p s v main(String[] args){
                Sopln(Example.a);
                Sopln(Example.b);
        }
}

class Example
{
        static int a = 15;
        static int b = 16;
}
```

Output

```
>javac Sample.java
>java Sample
15
16
```

Case #4: From the below code how many class files are generated after compiling Sample.java.

Example.java

```
class Example
{
        static int a = 70;
        static int b = 80;
}

class A{}
class B{}
```

Sample.java

```
class Sample{
        p s v main(String[] args){
                Sopln(Example.a);
                Sopln(Example.b);
        }
}
```

Output

```
>javac Sample.java
        |->A.java
        |->B.java
        |->Example.java
        |->Sample.java

4 class files are generated
```

Case #5: Change Example class name to Test in Example.java file, delete Example class definition in Sample.java file, also Example.class from current working directory. Then compile and execute Sample.java file, now you will get CE: cannot find symbol.

Example.java

```
class Test
{
        static int a = 70;
        static int b = 80;
}
```

Sample.java

```
class Sample{
        p s v main(String[] args){
                Sopln(Example.a);
                Sopln(Example.b);
        }
}
```

Output

```
>javac Sample.java
CE: cannot find symbol
Symbol: variable Example
location: class Sample
```

Case #6: Delete Example.java file and now compile Sample.java, in this case also you will get same above compile time error.

Creating sub packages:

Syntax:
pacakge parentpackagename.subpackagename;

For example: package p1.p2;

Below program shows creating sub package
//Example.java

```
package p1.p2;

class Example{
        public static void main(String[] args) {
                System.out.println("In subpackage");
        }
}
```

Compilation
>javac -d . Example.java

Execution
>java p1.p2.Example
In subpackage

Can we create classes with predefined class name?
Yes, we can create user defined classes or custom classes with predefined class name.

Then how can we differentiate these two classes?
By using package name If there is any class defined locally with same predefined class name, we must access predefined class name with its package name. Else it is loaded from current working directory if Classpath environment variable is setup with "." operator.

If there is a class with name "String" in your current working directory, will the classes defined in that directory compiled and executed?
Programs are compiled but will not be executed. Because JVM consider main method parameter is current local String class not predefined class. To solve this problem we must access main method parameter with package name "java.lang"

What is the output from below program?

```
class String{
        public static void main(java.lang.String[] args){
                java.lang.String String = "abc";
                System.out.println(String);
        }
}
```

Q) How can we access other package classes from our package classes?
There are two ways

1. By using fully qualified name or
2. By using import keyword

Understanding "import" keyword

import keyword is used to "access" other package members from this package classes.

Actually it does not import other package members into this package; instead it shows the path of the other package member to compiler and JVM.

We have 2 syntaxes to use import statement

```
syntax:
        import packagename.*;
                or
        import packagename.classname;
```

```
For example
        import p1.*;
                or
        import p1.Example;
```

What is the difference between above two import syntaxes?

First syntax allows Compiler & JVM to access all public members (classes, interfaces & enums) of that imported package, where as second syntax allows Compiler and JVM to access only that imported class.

Rule: import statement must be placed before all class definitions, and after package statement.

How many import statements are allowed in one Java file?

In a Java file "more than one import" statements and "only one package" statement are allowed.

Rule: To access packaged members from another package class its members must be declared as public, else it leads to compile time error while access that member.

Below program shows accessing other package members

```
package p1;
public class A{
        public static void m1(){
                System.out.println("A m1");
        }
}
```

Case #1: If user class is also defined in same package, import statement and fully qualified name is optional.

```
pacakge p1;
class B{
        public static void main(String[] args) {
                System.out.println("B main");
                A.m1();
        }
}
```

Output:

```
>javac -d . A.java
>javac -d . B.java
>java p1.B
B main
A m1
```

Case #2: If the class is using from another package, either import or fully qualified name must be used. Else it leads to CE: cannot find symbol.

```
package p2;
class C{
        public static void main(String[] args) {
                System.out.println("C main");
                A.m1(); //CE: cannot find symbol Class A
        }
}
```

```
package p2;
import p1.*;
class C{
        public static void main(String[] args) {
                System.out.println("C main");
                A.m1();
        }
}
```

Output:

```
>javac -d . C.java
>java p2.C
C main
A m1
```

What is the output from below program? Is there any compile time errors?

```
//A.java – class with
                default no-arg constructor
package p1;
public class A{
        public void m1(){
                System.out.println("A m1");
        }
}
```

```
//B.java – class with
                non-public non-arg constructor
package p1;
public class B{
        B(){
                System.out.println("B constructor");
        }
        public void m1(){
                System.out.println("B m1");
        }
}
```

```
//C.java – class with
        public parameterized constructor
package p1;
public class C{
        private C(){
           Solln("C no-arg constructor");
        }
        public C(String s){
           Sopln("C String constructor");
        }
        public void m1(){
                Sopln("C m1");
        }
}
```

```
//Test.java
package p2;
import p1.*;

public class Test{
        public static void main(String[] args){
                A a = new A();
                B b = new B();
                C c1 = new C();
                C c2 = new C("abc");
        }
}
```

```
//Test.java
package p2;
import p1.X;

public class Test{
        public static void main(String[] args){
                A a = new A();
                B b = new B();
                C c1 = new C();
                C c2 = new C("abc");
        }
}
```

What is the benefit we get in using import statement over fully qualified name?

If we do not use import statement, we must use package name every wherever we are using class name or constructor. This code is considered as redundant code and also code is not readable. To solve this problem SUN introduced "import" concept So, if we use import statement we no need to use package name in referring class name and its constructor.

Below program shows above problem

```
package p2;
class Sa{
        public static void main(String[] args) {
                p1.A a1 = new p1.A();
                p1.A a2 = new p1.A();

                p1.C c1 = new p1.C("a");
                p1.C c2 = new p1.C("b");
        }
}
```

Now this code is not readable.

Below program is the conversion program with import – we used package name only once

```
package p2;
import p1.*;

class Sa{
        public static void main(String[] args) {
                A a1 = new A();
                A a2 = new A();

                C c1 = new C("a");
                C c2 = new C("b");
        }
}
```

Now this code is readable.

Give a scenario where both import and fully qualified name should be used?
If a class with same name is available in two packages, to that class from both packages we must use fully qualified name to differentiate class from another packaged class. In this case if we use just import statements compiler throws ambiguous error in accessing that class.

Below program shows above CE

//A.java

```
package p3;
public class A{
        public A (){
            Soln("p3.A constructor");
        }
}
```

//Test.java

```
package p2;
import p1.*;
import p3.*;

public class Test{
        public static void main(String[] args){
                //A a = new A(); CE:
                p1.A a = new p1.A();
                p3.A a = new p3.A();
                C c = new C("abc");
        }
}
```

Using sub package members
we must import sub package separately to access its members. Because by importing parent package sub package members are not imported vice versa is also not possible.

Why sub package members are not imported, when we import parent package?
Because parent package may have more than one sub package, so compiler and JVM cannot take decision from which sub package that class must be accessed.

//D.java

```
package p1.p4;
public class D{
        public void m1(){  Sopln("D m1" ); }
}
```

Find out CEs in the below programs

//Test.java

```
package p2;
import p1.*;

class Test{
        public static void main(String[] args) {
                A a = new A();
                D d = new D();
        }
}
```

//Test.java

```
package p2;
import p1.*;
import p1.p4.*;

class Test{
        public static void main(String[] args) {
                A a = new A();
                D d = new D();
        }
}
```

Static imports:
This feature is introduced in Java 5 to import static members of a class.

By using this feature we can access all

- non-private static members without using class name from other classes with in the package and
- protected and public members from outside package class members without using class name.

Syntax:

```
import static packagename.classname.*;
                    or
import static packagename.classname.staticmembername;
```

- first syntax allows to call all static members of the class.
- second syntax allows only to call the imported static member.

Q) What is the difference between below statements?

import p1.*;
- We can access all classes from p1 package

import p1.A;
- We can access only class A from p1 package

import static p1.A.*;
- We can access all static members of class A from p1 package
- Using this import statement we cannot access non-static members, we cannot create object, we cannot develop subclass from A class.
- for this purpose we must also write import statement separately for accessing class A as "import p1.A;"

import static p1.A.a;
- We can only access the static variable "a"
- if "a" is a non-static variable it leads to CE: cannot find symbol "static a"

import static p1.A.m1;
- We can only access the static method "m1"
- if "m1" is a non-static method it leads to CE: cannot find symbol "static m1"

Below program shows accessing static members of a class with "static import" concept.

//Example.java

```
package p1;
public class  Example {
        public static int a = 10;
        public int x = 20;

        public static void m1(){
                Sopln("m1");
        }
        public void m2(){
                Sopln("m2");
        }
}
```

//Sample.java

```
package p2;
import static p1.Example.*;

public class Sample {
   public static void main(String[] args){
        //accessing static members without using classname
        System.out.println(a);
        m1();

        //accessing static members with classname
        System.out.println(Example.a); //CE:
        Example.m1(); //CE:

        //accessing non-static members
        Example e = new Example(); CE:
        System.out.println(e.x);
        e.m2();
   }
}
```

Note: To solve above compile time errors we must also import class Example with normal import statement as "import p1.Example;"

Q) If the current class also contains the imported static member, how can we differentiate both of them? We must use fully qualified name of the that static member that is "packagename.classname.staticmembername" else current class static member is used.

Check below program

//Sample.java

```
package p2;
import static p1.Example.*;
class Sample{
        static int a = 70;

        public static void main(String[] args){
                System.out.println(a);
                //System.out.println(Example.a); //CE: cfs varaible Example
                System.out.println(p1.Example.a);

                m1();
        }
}
```

Q) Write a program to print data without using class name System.
You should use only "out.println()" to print "hi", "hello", "hru?"

//Test.java

```
package p2;
import static java.lang.System.*;
class Test{
        public static void main(String[] args){
                out.println("Hi");
                out.println("Hello");
                out.println("Hru?");
        }
}
```

Find out valid syntaxes from the below list

```
import java.lang.*;
import java.lang.System;
import java.lang.System.*;
import java.lang.System.out;

import static java.lang.System.*;
import static java.lang.System;
import static java.lang.System.out;
import static java.lang.System.out.*;
static import java.lang.System.out;
```

Q) In the below program at what line number CE is raised?

//SISyntax2.java

```
1. import p1.A.*;

2. class SISyntax2{
3.      public static void main(String[] args){
4.              m1();
5.              A.m1();
6.              p1.A.m1();
7.      }
8. }
```

Java source file structure

In a java file we can have package statement, import statement, interface, abstract class, concrete class, final class, main method class, and documentation.

All these are organized as shown below according to coding standards and compiler rules

Documentation section
Package statement
Import statement
Interface
Abstract class
Concrete class
Final class
Main method class

In-built packages

- SUN given packages are called in-built packages, and developer given packages are called custom or user defined packages.
- SUN also organized all predefined classes, interfaces and enums in packages.
- We have two root packages for in-built packages
- They are
  - java and javax

- java package has basic and fundamental or core classes and interface for design of java programming language.
- javax package has extension classes and interfaces.
  - javax stands for "Java eXtension"
- Below diagram shows the Java SE important sub packages of *java* and *javax* packages

java
- |- lang
- |- io
- |- net
- |- util
- |- awt
- |- applet
- |- sql
- |- rmi
- |- math
- |- text

javax
- |- swing
- |- sql
- |- xml
- |- naming
- |- transaction