Day7: Access Modifiers, Packages, Abstract methods and Abstract class

Packages in Java:

Package in **Java** is a mechanism to encapsulate a group of classes, sub-packages and interfaces.

Packages are used for:

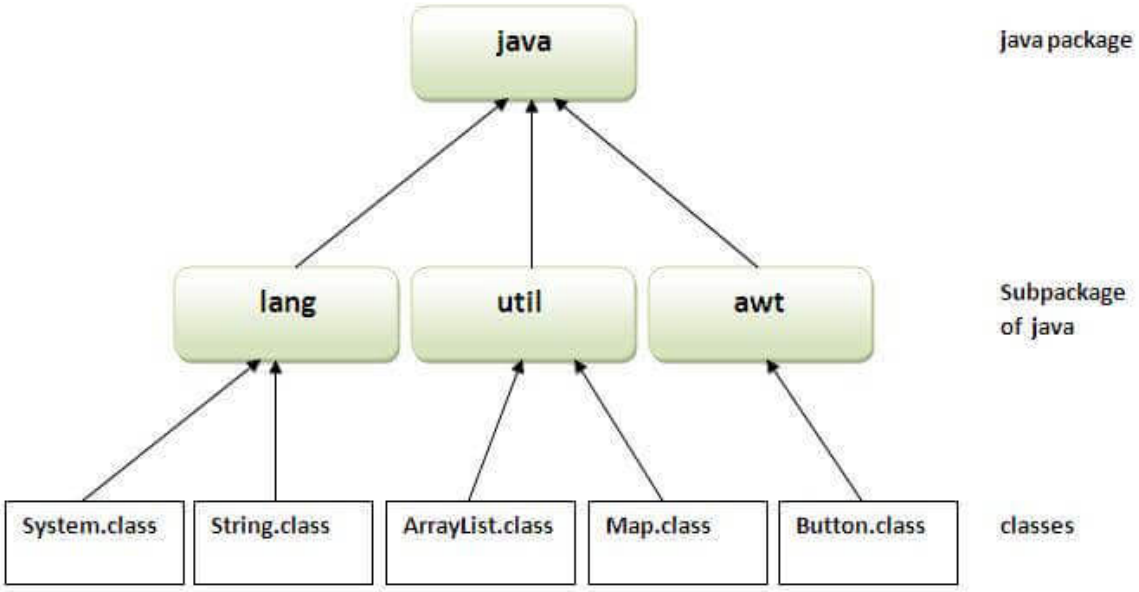
- Preventing naming conflicts. For example, there can be two classes with the name Employee in two packages, college.staff.cse.Employee and college.staff.ee.Employee
- Group the related classes, interfaces, etc logically. so that they can be easily maintained.
- Providing controlled access: protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.

Some of the existing packages in Java are −

- **java.lang** package− bundles the fundamental classes
- **java.io** package − classes for input, output functions are bundled in this package

Note: A Package is like **a folder in a file directory,** In Java, every package is a folder but every folder is not a package.

.



If a package name is ***college.staff.cse***, then there are three directories, *college*, *staff* and *cse* such that *cse* is present in *staff* and *staff* is present *college.*

Package naming conventions :

Packages are named in reverse order of domain names, i.e., com.masai;

For example, in a college, the recommended convention is college.tech.cse, college.tech.ee, college.art.history, etc.

Accessing classes from a package:

If a java class belongs to a package, in order to use that class inside our java application (or inside a different class) we need to import that class .

Example:

```
// import the Scanner class from the util package.
import java.util.Scanner;

// import all the classes from util package
import java.util.*;
```

```
import java.util.*;

It will import all the classes and interfaces from java.util package only
not from any sub-packages, like "java.util.function"
```

User-defined Packages:

To create a package, use the `package` keyword:

example

```
/save as Simple.java
package mypack;
public class Simple{
   public static void main(String args[]){
      System.out.println("Welcome to package");
   }
}
```

.

How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

```
javac -d . fileName.java
```

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

How to run a java package program

You need to use fully qualified name i.e packageName.className

example:

```
>java mypack.Simple
```

Creating sub-packages:

Let's take an example, Sun Microsystem has defined a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc. and so on. So, Sun has subcategorized the java package into sub-packages such as lang, net, io etc. and put the Input/Output related classes in the io package, Server and ServerSocket classes in net packages and so on.

Example of Sub-package:

```
package com.masai.core;
class Simple{
```

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

Java Static Import [not recommended to use]

The static import feature of Java 5 facilitates the java programmer to access any static member of a class directly. There is no need to qualify it by the class name.
For Example: we always use sqrt() method of Math class by using Math class
i.e. **Math.sqrt()**
but by using static import we can access sqrt() method directly.

Advantage of static import:

- Less coding is required if you have access to any static member of a class often.

The Disadvantage of static import:

- If you overuse the static import feature, it makes the program unreadable and unmaintainable.

Example:

```
package com.masai;
import static java.lang.System.*;
public class Demo{
  public static void main(String args[]){

   out.println("Hello");//Now no need of System.out
   out.println("Java");

 }
}
```

Access Modifier In Java:

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

Access modifier table

There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. Aka package-protected. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Note: An outer class can only be default or public, where as class members can be public, private, protected, default.

Role of Private Constructor:

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

```
class A{
  private A(){//private constructor
  }

  void msg(){
```

```
    System.out.println("Hello java");
  }
}
public class Simple{
 public static void main(String args[]){
   A obj=new A();//Compile Time Error
 }
}
```

Example of default access modifier:

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
//save by A.java
package pack;
class A{
  void msg(){
    System.out.println("Hello");
  }
}

//save by B.java
package mypack;
import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();//Compile Time Error
   obj.msg();//Compile Time Error
  }
}
```

protected:

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

It provides more accessibility than the default modifer.

Example:

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
//save by A.java
package pack;
public class A{
protected void msg(){
  System.out.println("Hello");
}
}

//save by B.java
package mypack;
import pack.*;

class B extends A{
  public static void main(String args[]){
   B obj = new B();
   obj.msg();
  }
}
```

Java Access Modifiers with Method Overriding:

If you are overriding any method, (declared in subclass) must not be more restrictive.

Example:

```
class A{
 public void  msg(){
```

```
    System.out.println("Hello java");
}
}

class Simple extends A{

 void msg(){
  System.out.println("Hello java");
  }//C.T.Error

public static void main(String args[]){
   Simple obj=new Simple();
   obj.msg();
   }
}
```

Abstraction in Java:

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

There are two ways to achieve abstraction in java:

1. Abstract class (0 to 100%)
2. Interface (100%): Since Java 8, we have default methods.

Abstract Class:

A class that is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Important points of Abstract class:

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated (object creation).
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

Abstract Method In Java:

Sometimes, we require just method declaration in super-classes. This can be achieve by specifying the **abstract** type modifier. These methods are sometimes referred to as *sub-classer responsibility*
because they have no implementation specified in the super-class. Thus, a subclass must **override**
them to provide method definition. To declare an abstract method, use this general form:

```
abstract type method-name(parameter-list);
```

As you can see, no method body is present. Any concrete class(i.e. class without abstract keyword) that extends an abstract class must override all the abstract methods of the class.

Important rules for abstract methods:

1. Abstract methods don’t have the body, they just have method signature.

2. If a class has an abstract method it should be declared abstract, the vice versa is not true, which means an abstract class doesn’t need to have an abstract method compulsory.

3. If a regular class extends an abstract class, then the class must have to implement all the abstract methods of abstract parent class or it has to be declared abstract as well.

4. Any class that contains one or more abstract methods must also be declared abstract
5. The following are various **illegal combinations** of other modifiers for methods with respect to *abstract* modifier:
    a. final - abstract
    b. abstract static
    c. abstract private

I Problem:

Creating an abstract class Bike with an abstract method run, after that creating a concrete class Apache as a child class of Bike and implementing the run method.

```
//Bike.java:

abstract class Bike{
  abstract void run();
}

//Apache.java:

class Apache extends Bike{
  void run(){
      System.out.println("running safely");
  }

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

```
 Bike bike1 = new Apache();
  bike1.run();
}
}
```

We Problem:

Lets create an abstract class Shape with an abstract method draw. and create 2 concreate child classes for this Shape class Rectangle and Circle and override the draw method accordingly.

```
//Shape.java

abstract class Shape{
  abstract void draw();
}

//Rectangle.java
class Rectangle extends Shape{
  void draw(){
    System.out.println("drawing rectangle");
  }
}

//Circle.java
class Circle extends Shape{
  void draw(){
    System.out.println("drawing circle");
  }
}

//Main.java

class Main{
public static void main(String args[]){

  Shape shape1=new Rectangle();
  Shape shape2=new Circle();

  shape1.draw();
  shape2.draw();
```

```
}
}
```

You Problem:

1. Create an abstract class Bank with an abstract method

   abstract int getRateOfInterest();
2. create 3 concreate child classes of this Bank class SBI, PNB, ICICI

   override the getRateOfInterest() method and return the values as follows:

   SBI : rateOfInterest = 7;

   PNB : rateOfInterest = 8;

   ICICI : rateOfInterest = 9;
3. create a BankDemo class with the main method and call this rateOfInterest() method

   on all three SBI, PNB, ICICI objects with Bank class reference.

   for example:

   Bank bank1 = new PNB();

   System.out.println("Rate of Interest of PNB is: "+bank1.getRateOfInterest()+" %");

   An abstract class can have a data member, abstract method, method body (non-abstract method), constructor, and even main() method.

   Example:

```
//Bike.java
abstract class Bike{

    Bike(){ //constructor
      System.out.println("bike is created");
    }
```

```
   abstract void run();

  void changeGear(){
      System.out.println("gear changed");
  }

}

//Creating a Child class which inherits Abstract class
//Honda.java
 class Honda extends Bike{
   void run(){
      System.out.println("running safely..");
    }
 }

//Creating a Main class which calls abstract and non-abstract methods
  //Main.java
 class Main{
   public static void main(String args[]){

      Bike obj = new Honda();
      obj.run();
      obj.changeGear();
 }
}

Output:
       bike is created
       running safely..
       gear changed
```

Rules:

1. **If there is an abstract method in a class, that class must be abstract.**
2. **If you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make this class abstract.**

Reference:

https://www.w3schools.com/java/java_packages.asp

https://www.javatpoint.com/

https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html