UNIT - II

Unit - II Syllabus:

Inheritance: Basic concepts, access specifiers, usage of super key word, method overriding, using final with Inheritance, abstract classes, dynamic method dispatch, Object class.
Interfaces: Differences between classes and interfaces, defining an interface, implementing interface, variables in interface and extending interfaces.
Packages: Creating a Package, setting CLASSPATH, Access control protection, importing packages.
Strings: Exploring the String class, String buffer class, Command-line arguments.

Inheritance:

Basic Concepts:

Inheritance is the mechanism of deriving new class from old one, old class is knows as superclass and new class is known as subclass.

The subclass inherits all of its instances variables and methods defined by the superclass and it also adds its own unique elements.

Benefits of Java's Inheritance

Reusability of code

Code Sharing

Superclass(Base Class)

It is a class from which other classes can be derived.

Subclass(Child Class)

It is a class that inherits some or all members from superclass.

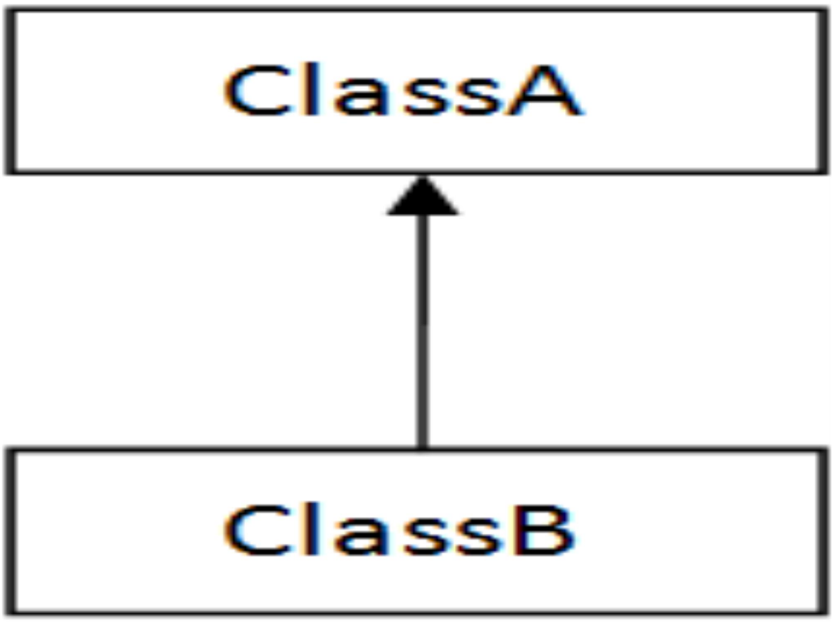
Syntax:

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

Types of Inheritance in Java:

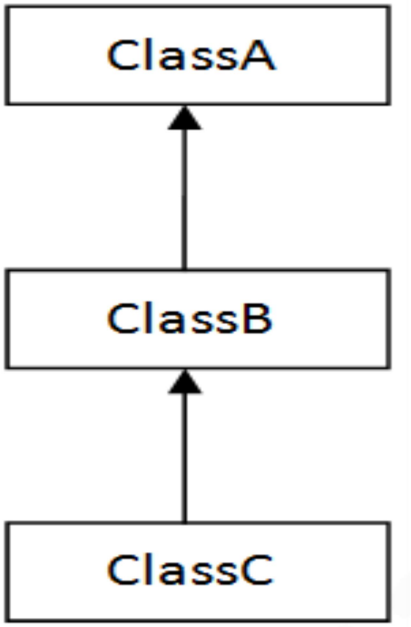
Single Inheritance:

When a class inherits another class, it is known as a single inheritance.



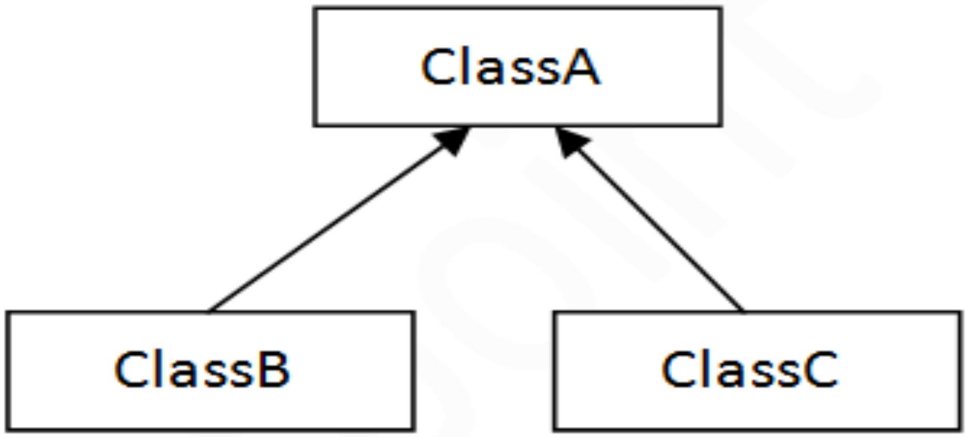
Multilevel Inheritance:

When there is a chain of inheritance, it is known as multilevel inheritance.



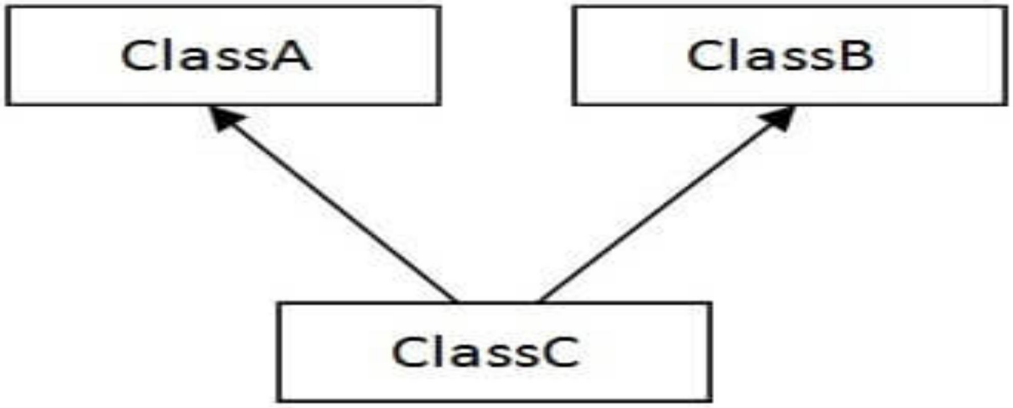
Hierarchical Inheritance:

When two or more classes inherits a single class, it is known as hierarchical inheritance.



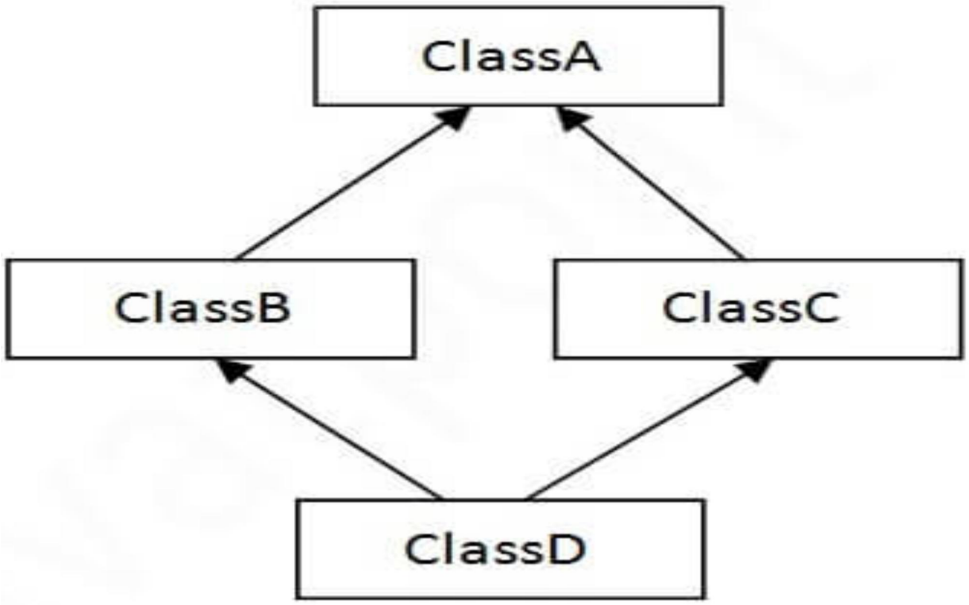
Multiple Inheritance:

Multiple Inheritance" refers to the concept of one class extending (Or inherits) more than one base class.



Hybrid Inheritance:

Hybrid inheritance is a combination of Single and Multiple inheritance.



Super Keyword:

The super keyword in Java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword:

1. super can be used to refer immediate parent class instance variable.

We can use super keyword to access the data member or field of parent class.

It is used if parent class and child class have same fields.

Example:

```
class Animal{
String color="white";
}
class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}
}
```

2. super can be used to invoke immediate parent class method.

The super keyword can also be used to invoke parent class method.

It should be used if subclass contains the same method as parent class.

In other words, it is used if method is overridden.

Example:

```
class Animal{
void eat(){System.out.println("eating...");}
```

```
}
class Dog extends Animal{
void eat(){System.out.println("eating bread...");}
void bark(){System.out.println("barking...");}
void work(){
super.eat();
bark();
}
}
```

3. super() can be used to invoke immediate parent class constructor.

The super keyword can also be used to invoke the parent class constructor.

Example:

```
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
super();
System.out.println("dog is created");
}
}
```

Method Overriding:

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass.

When an overridden method is called from within its subclass, it will always refer to the version of that method defined by the subclass.

The version of the method defined by the superclass will be hidden.

Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.

A static method cannot be overridden.

Example:

```
class Vehicle{
  //defining a method
  void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike2 extends Vehicle{
  //defining the same method as in the parent class
  void run(){System.out.println("Bike is running safely");}
}
```

Output:

Bike is running safely.

Final keyword with inheritance in java:

The final keyword is final that is we cannot change.

We can use final keywords for variables, methods, and class.

If we use the final keyword for the inheritance that is if we declare any method with the final keyword in the base class so the implementation of the final method will be the same as in derived class.

We can declare the final method in any subclass for which we want that if any other class extends this subclass.

Example:

```
class Parent {
      final String pa = "Hello , We are in parent class variable";
}
class Child extends Parent {
      String ch = "Hello , We are in child class variable";
}

class Test {
      public static void main(String[] args) {
            Parent p = new Parent();
            System.out.println(p.pa);
            Child c = new Child();
            System.out.println(c.ch);
            System.out.println(c.pa);
      }
}
```

Output:

```
Hello , We are in parent class variable
Hello , We are in child class variable
Hello , We are in parent class variable
```

Abstract Classes:

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

There are two ways to achieve abstraction in java

1. Abstract class
2. Interface

A class which 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.

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

A method which is declared as abstract and does not have implementation is known as an abstract method.

Example:

```
abstract class Bike{
  abstract void run();
}
class Honda4 extends Bike{
void run(){System.out.println("running safely");}
public static void main(String args[]){
 Bike obj = new Honda4();
 obj.run();
}
}
```

If there is an abstract method in a class, that class must be abstract.

```
class Bike12{
        abstract void run();
}
```

If we are extending an abstract class that has an abstract method, we must either provide the implementation of the method or make this class abstract.

Dynamic Method Dispatch:

Runtime polymorphism or Dynamic Method Dispatch is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Example:

```
class Shape{
void draw(){System.out.println("drawing...");}
}
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle...");}
}
class Circle extends Shape{
void draw(){System.out.println("drawing circle...");}
}
class Triangle extends Shape{
void draw(){System.out.println("drawing triangle...");}
}
class TestPolymorphism2{
public static void main(String args[]){
```

```
Shape s;
s=new Rectangle();
s.draw();
s=new Circle();
s.draw();
s=new Triangle();
s.draw();
}
}
```

Output:

```
drawing rectangle...
drawing circle...
drawing triangle...
```

Difference between method overloading and method overriding

No.	Method Overloading	Method Overriding
1)	Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
2)	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

Object Class:

The Object class is the parent class of all the classes in java by default. In other words, it is the topmost class of java.

Every class in Java is directly or indirectly derived from the Object class.

The Object class is beneficial if you want to refer any object whose type you don't know.

If a Class does not extend any other class then it is direct child class of Object and if extends other class then it is an indirectly derived.

Object class is present in java.lang package

Object defines the following methods, which means that they are available in every object.

Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object *object*)	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled.
Class<?> getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( ) void wait(long *milliseconds*) void wait(long *milliseconds*, int *nanoseconds*)	Waits on another thread of execution.

Interfaces:

Differences between classes and interfaces

Class	Interface
A class describes the attributes and behaviour of an object.	An interface contains behaviour that a class implements.
A class may contain abstract methods, concrete methods.	An interface contains only abstract methods.
Members of a class can be public, private, protected or default.	All the members of the interface are public by default.
A class can be instantiated	An interface can never be instantiated
The **class** keyword is used to declare it	The **interface** keyword is used
The **extends** keyword is used to inherit a class	The **implements** keyword is used to use an interface
A Java class can have constructors	An interface cannot have constructors
A class can extend only one class but can implement any number of interfaces	An interface can extend any number of interfaces but cannot implement any interface

Defining an Interface:

An interface in Java is a blueprint of a class.
It has static constants and abstract methods.
It is used to achieve abstraction and multiple inheritance in Java.
Interfaces can have abstract methods and variables. It cannot have a method body.
It cannot be instantiated just like the abstract class.

Syntax:

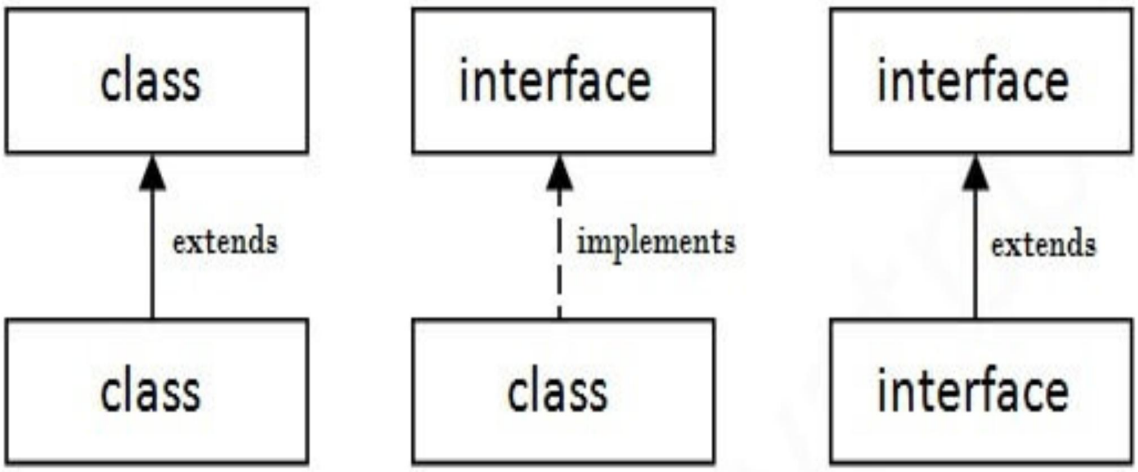
```
interface <interface_name>{

  // declare constant fields
  // declare methods that abstract
  // by default.
}
```

An interface is declared by using the interface keyword.
It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default.
A class that implements an interface must implement all the methods declared in the interface.

The relationship between classes and interfaces

A class extends another class, an interface extends another interface, but a class implements an interface.



Example:

```
interface printable{
void print();
}
class A6 implements printable{
public void print(){System.out.println("Hello");}

public static void main(String args[]){
A6 obj = new A6();
obj.print();
 }
}
```

Output:

Hello

Implementing an Interface:

Multiple Inheritance:

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.

Multiple Inheritance in Java

Multiple inheritance is not supported in the case of class because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class.

Example:

```
interface Printable{
void print();
}
interface Showable{
void print();
}
class TestInterface3 implements Printable, Showable{
public void print(){System.out.println("Hello");}
public static void main(String args[]){
TestInterface3 obj = new TestInterface3();
obj.print();
 }
}
```

Output:

Hello

Interface Inheritance:

A class implements an interface, but one interface extends another interface.

Example

```
interface Printable{
void print();
}
interface Showable extends Printable{
void show();
}
class TestInterface4 implements Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

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

```
TestInterface4 obj = new TestInterface4();
obj.print();
obj.show();
 }
}
```

Output:

```
Hello
Welcome
```

Interface Variables:

We can declare variables in Java interfaces.
By default, these are public, final and static.
That is they are available at all places of the program, we can not change these values and lastly only one instance of these variables is created, it means all classes which implement this interface have only one copy of these variables in the memory.

Example:

```
interface X {
	int max = 10;
}

class Example implements X {
	public void getMax()
	{
		System.out.println(max);
	}
}

class KH_InterfaceVariables {
	public static void main(String args[]) {
		Example ob = new Example();

		ob.getMax();
		    //ob.max=20
		}
	}
```

Extending Interfaces:

Nested Interface:

An interface i.e. declared within another interface or class is known as nested interface.

Syntax

```
interface interface_name{
```

```
 ...
 interface nested_interface_name{
  ...
 }
}
```

(OR)

```
class class_name{
 ...
 interface nested_interface_name{
  ...
 }
}
```

Example:

```
interface Showable{
  void show();
  interface Message{
   void msg();
  }
}
class TestNestedInterface1 implements Showable.Message{
 public void msg(){System.out.println("Hello nested interface");}

 public static void main(String args[]){
  Showable.Message message=new TestNestedInterface1();//upcasting here
  message.msg();
 }
}
```

Output:

Hello nested interface

Example of nested interface which is declared within the class

```
class A{
  interface Message{
   void msg();
  }
}
  class TestNestedInterface2 implements A.Message{
 public void msg(){System.out.println("Hello nested interface");}

 public static void main(String args[]){
  A.Message message=new TestNestedInterface2();//upcasting here
  message.msg();
 }
}
```

Output:

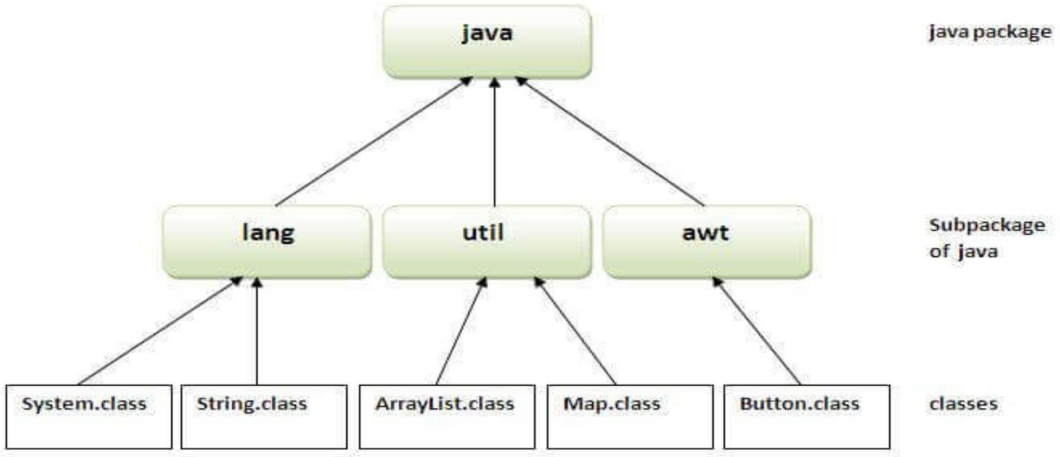
Hello nested interface

Packages

A java package is a group of similar types of classes, interfaces and sub-packages.
Package in java can be categorized in two form, built-in package and user-defined package.
There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Advantage of Java Package

Java package is used to categorize the classes and interfaces so that they can be easily maintained.
Java package provides access protection.
Java package removes naming collision.



Creating a Package:

To create a package is quite easy: simply include a package command as the first statement in a Java source file.
Any classes declared within that file will belong to the specified package.
The package statement defines a name space in which classes are stored.
If we omit the package statement, the class names are put into the default package, which has no name.

Syntax:

package pkg;

pkg is the name of the package.

compiling java package

javac -d directory javafilename

Example

javac -d . Simple.java

The -d switch specifies the destination where to put the generated class file.

We can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc.

If we want to keep the package within the same directory, we use . (dot).

setting CLASSPATH:

Packages are mirrored by directories.

How does the Java run-time system know where to look for packages that we create?

First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found.

Second, we can specify a directory path or paths by setting the CLASSPATH environmental variable.

Third, you can use the -classpath option with java and javac to specify the path to our classes.

Example

package MyPack

In order for a program to find MyPack, one of three things must be true. Either the program can be executed from a directory immediately above MyPack, or the CLASSPATH must be set to include the path to MyPack, or the -classpath option must specify the path to MyPack when the program is run via java.

When the second two options are used, the class path must not include MyPack, itself. It must simply specify the path to MyPack. For example, in a Windows environment, if the path to MyPack is C:\MyPrograms\Java\MyPack then the class path to MyPack is C:\MyPrograms\Java

Example:

```
package MyPack;
class Balance {
String name;
double bal;
Balance(String n, double b) {
name = n;
bal = b;
}
void show() {
if(bal<0)
System.out.print("--> ");
System.out.println(name + ": $" + bal);
}
}
class AccountBalance {
public static void main(String args[]) {
Balance current[] = new Balance[3];
current[0] = new Balance("K. J. Fielding", 123.23);
current[1] = new Balance("Will Tell", 157.02);
```

```
current[2] = new Balance("Tom Jackson", -12.33);
for(int i=0; i<3; i++) current[i].show();
}
}
```

Call this file AccountBalance.java and put it in a directory called MyPack.

Next, compile the file. Make sure that the resulting .class file is also in the MyPack directory.

Try to executing the AccountBalance class, using the following command line:

```
java MyPack.AccountBalance
```

Remember, you will need to be in the directory above MyPack when you execute this command. (Alternatively, we can use one of the other two options

AccountBalance is now part of the package MyPack. This means that it cannot be executed by itself. So we cannot use this command line:

```
java AccountBalance
```

AccountBalance must be qualified with its package name.

Access Control Protection:

Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages.

Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods.

Packages act as containers for classes and other subordinate packages.

Classes act as containers for data and code. The class is Java's smallest unit of abstraction.

Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:

Subclasses in the same package

Non-subclasses in the same package

Subclasses in different packages

Classes that are neither in the same package nor subclasses

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Exaample(s):
https://ecomputernotes.com/java/packages/access-protection-in-packages

Importing Packages(Lab Assignment 4):

Step 1: Create a folder named java on Desktop

Step 2: Create 2 folders(Packages and Source) inside java directory.

Step 3: set CLASSPATH=C:\Users\Yogi\Desktop\Java\Packages; C:\Program Files\Java\jdk-19\lib

We should set these 2 classpaths. One is our package location and other is lib directory of jdk.

Step 4: In source directory create a text document and write the shape2d,shape3d,Circle class code inside it and save it with Circle.java extension.

Make sure that you've given public keyword before circle class name,else it can't be imported.

Step 5: Open Command Prompt and cd(Change directory) to source folder

Step 6: Now use the following command:

javac-d ..\packages Circle.java

By using .. we move up one directory i.e outside of source and inside of java...Nd using \packages create the Shape package in packages directory.

Step 7: Now create another .java file named Cylinder inside source directory and write cylinder code inside the file. Cylinder class should be public.

Step 8: Now use the following command:

javac -d ..\packages Cylinder.java

By using .. we move up one directory i.e outside of source and inside of java...Nd using \packages create the Shape package in packages directory.

Step 9: Now Package Shape is ready.

Step 10: To test if the package works correctly, create a test.java file in source directory and write the code required.

Step 11: Compile it by using following command:

Javac Main.java followed by **java Main**

Strings

Strings PPT

***