MODULE II

BUILD CLASSES USING INHERITANCE ,INTERFACES AND PACKAGES

INHERITANCE

Inheritance in Java is a mechanism in which one class acquires all the properties and behaviors of a parent class. It is an important part of OOPs (Object Oriented programming system).The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance is also known as a **parent-child** relationship.

inheritance in java is used for the following:

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Terms used in Inheritance:

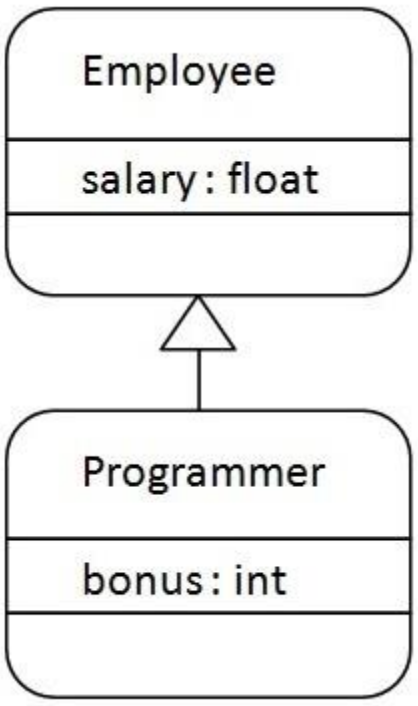
- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** Reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

The syntax of Java Inheritance

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

The **extends keyword** indicates that you are making a new class that derives from an existing class. A class which is inherited is called a **parent or superclass**, and the new class is called **child or subclass.**

java Inheritance Example



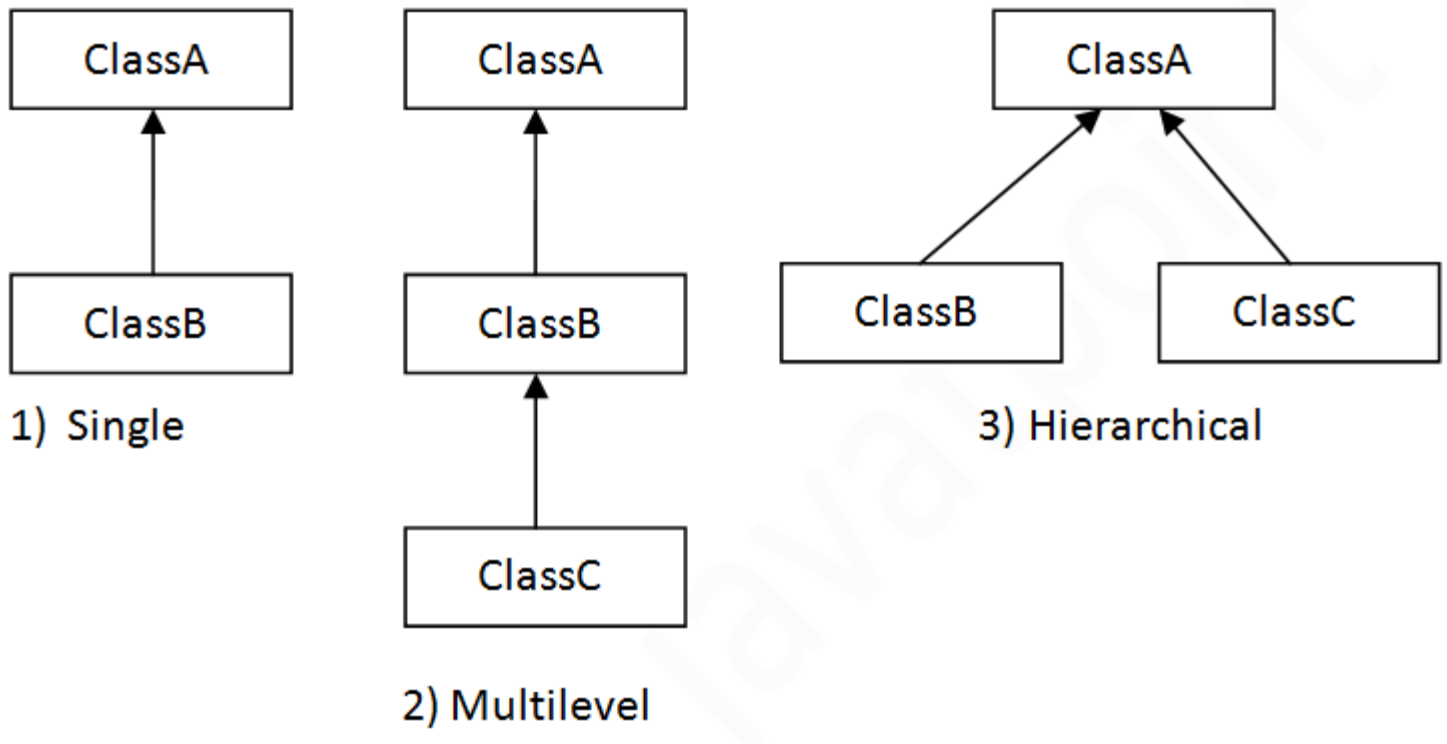
As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

Types of inheritance in java

On the basis of **class,** there can be three types of inheritance in java:

single, multilevel and hierarchical.

In java programming, **multiple** inheritance is supported through **interface** only.



(1)SINGLE INHERITANCE

When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

```
class Animal{
void eat(){
        System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){
        System.out.println("barking...");}
}
class TestInheritance{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}}
```

(2)MULTILEVEL INHERITANCE

When there is a chain of inheritance, it is known as *multilevel inheritance*.In the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

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

```
        System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
```

(3) HIERARCHICAL INHERITANCE:

When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
Dog d=new Dog();
c.meow();
c.eat();
d.bark();
d.eat();
}}
```

INTERFACES

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is *a mechanism to achieve* ***abstraction*****.** There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

In other words, you can say that interfaces can have **abstract methods** and variables. It cannot have a **method body**.

Java interface is used to achieve the following,

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.

Interface Declaration:

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.

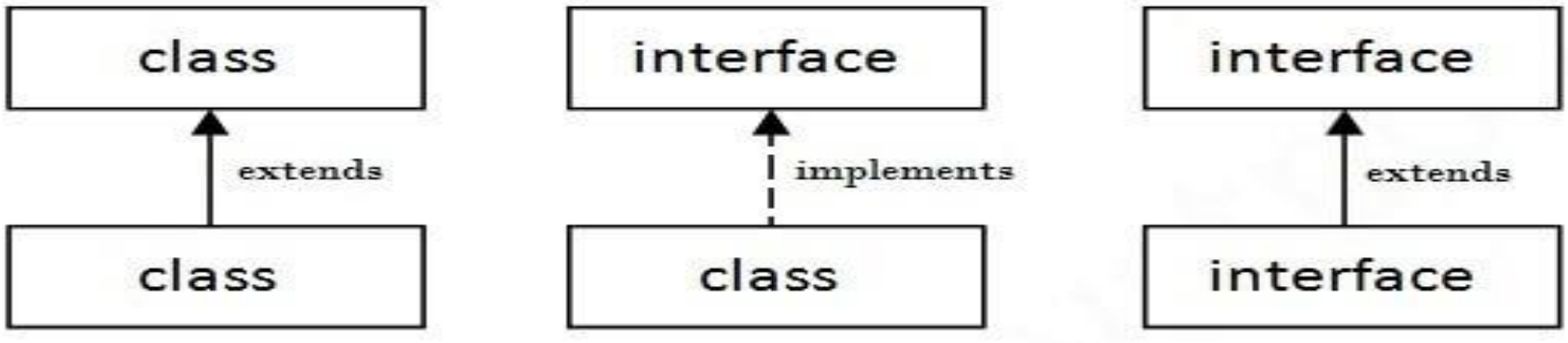
Syntax:

```
interface  interfacename{

        // declare constant fields
        // declare methods that abstract
        }
```

The relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



Java Interface Example

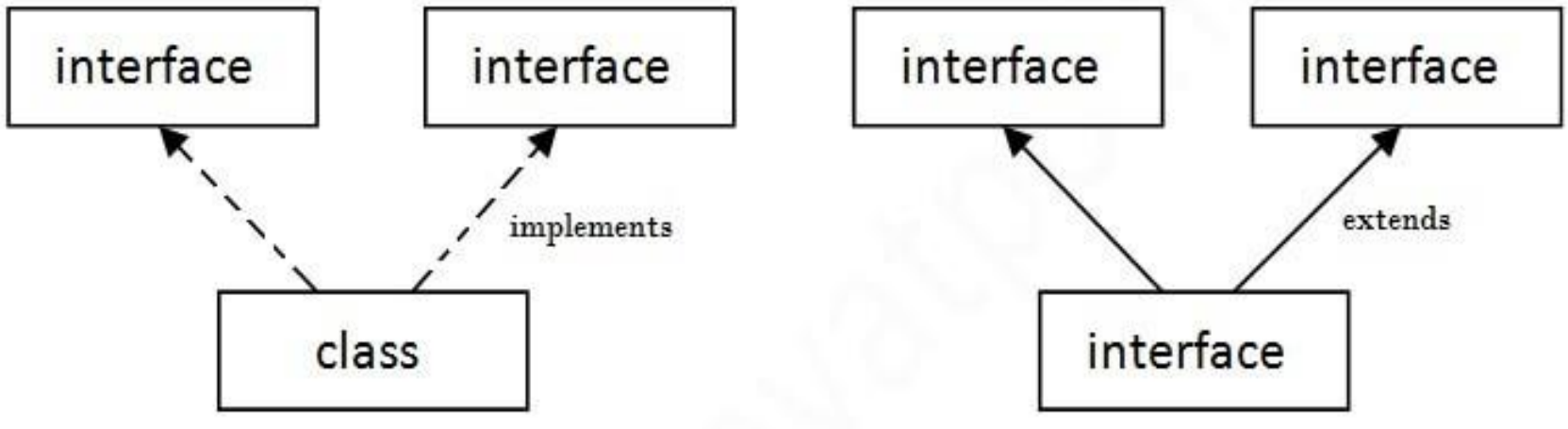
In this example, the Printable **interface** has only one method, and its implementation is provided in the class Test.

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

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

(4)MULTIPLE INHERITANCE in Java by interface

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



Multiple Inheritance in Java

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

FINAL VARIABLES ,FINAL METHODS AND FINAL CLASSES

Final Variables:

If you make any variable as final, you cannot change the value of final variable(It will be constant).

A final variable can be explicitly initialized only once. Set a variable to final, to prevent it from being overridden/modified:

The `final` keyword is useful when you want a variable to always store the same value, like PI (3.14...).

Example

```
public class Test {
   final int value = 10;
           void changeValue() {
      value = 12;   // will give an error
   }
}
```

Final Methods

If you make any method as final, you cannot override it.

A final method cannot be overridden by any subclasses. The final modifier prevents a method from being modified in a subclass.

The main intention of making a method final would be that the content of the method should not be changed by any outsider.

Example

```
    class Test {
           final void changeName() {
     // body of method
   }
}
```

Final Classes

If you make any class as final, you cannot extend it.

The main purpose of using a class being declared as *final* is to prevent the class from being subclassed. If a class is marked as final then no class can inherit any feature from the final class.

Example

```
  final class Test {
   // body of class
}
```

ABSTRACT CLASS AND ABSTRACT METHOD

Abstract Class:-

A class which is declared with the **abstract** keyword is known as an abstract class in Java. Abstract class can have **abstract and non-abstract methods**

Abstract method is method without the body.

Non-Abstract method is method with the body

Abstract class is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).

If a class contains **abstract methods** then the **class** should be declared **abstract**. Otherwise, a compile error will be thrown.

An abstract class may contain both abstract methods as well normal methods.

Example

```
abstract class Animal {
  public abstract void animalSound(); //abstact method
  public void sleep() {                 //normal method
    System.out.println("Zzz");
  }
}
```

Abstract Methods

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

Example of abstract method:

abstract void printStatus(); //no method body

An abstract method is a method declared without any implementation. The methods body (implementation) is provided by the subclass.

Any class that extends an abstract class must implement all the abstract methods of the super class,.If a class contains one or more abstract methods, then the class must be declared abstract.

The abstract method ends with a semicolon.

Example: public abstract void sample();

Abstract method: can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

Example :

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

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.

Ways to achieve Abstraction:

There are two ways to achieve abstraction in java

1. **Abstract class**
2. **Interface**

JAVA 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,

(1) **built-in package**
(2) **user-defined package.**

(1)BUILT- IN PACKAGES

There are many **built-in packages** such as **java, lang, awt, javax, swing, io, util, sql** etc. We can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.

To use a class or a package , you need to use the **`import`** keyword:

Syntax: `import package.name.Class;` `// Import a single class`

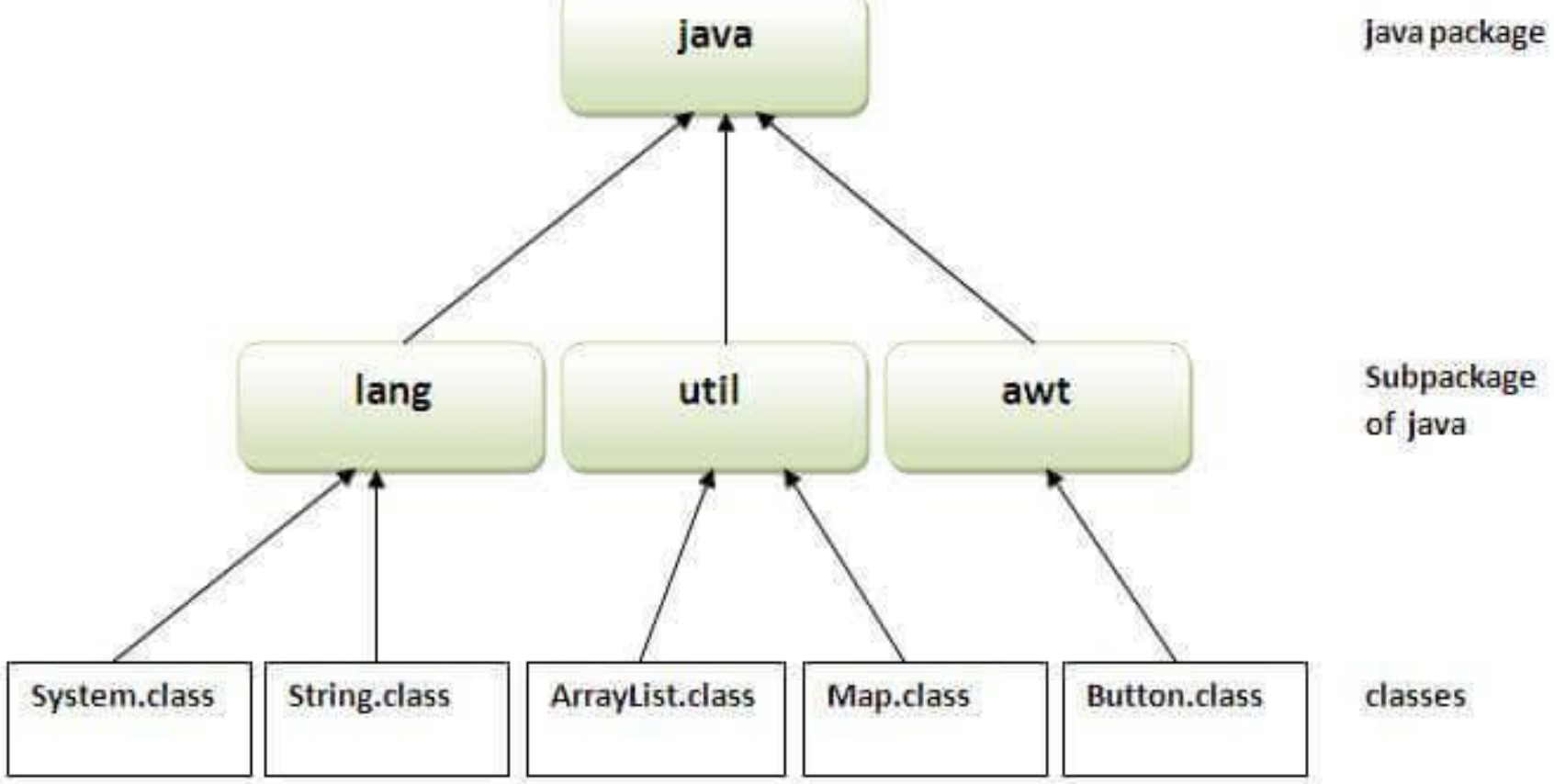
Eg: `import java.util.Scanner;`

Syntax: `Import package.name.*;` `// Import the whole package`

Eg: `import java.util.*;`

In the example above, **`java.util`** is a package, while **`Scanner`** is a class of the **`java.util`** package. To use the `Scanner` class, create an object of the class and use any of the available methods(`nextInt(),nextDouble()` `etc`) found in the `Scanner` class .

.



(2)USEREFINED package

The **package keyword** is used to create a package in java.

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;.

(1) Using packagename.*:

If you use **package.*** then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

(2) import package.classname;.

If you import package.**classname** then only declared class of this package will be accessible.

Example of userdefined packages:

Example1:

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

Example2:

(A)

```
package pack;
public class First{
   public void msg(){
            System.out.println("Hello");}
}
```

(B)

```
package mypack1;
import pack.*;
  class Second{
    public static void main(String args[]){
    First obj = new First();
   obj.msg() ;
              }
     }
```

NOTE: If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well. Package inside the package is called **subpackages**.

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 invoke **immediate parent class method**.
2. **super()** can be used to **invoke immediate parent class constructor**.

(1)super can be used to invoke 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.

```
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();
             }
              }
class TestSuper2{
public static void main(String args[]){
Dog d=new Dog();
d.work();
}}
```

(2) super is used to invoke 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");
          }
}
class TestSuper3{
public static void main(String args[]){
Dog d=new Dog();
}}
```

DYNAMIC BINDING

If you have more than one method of the same name (method overriding) in the same class it gets tricky to find out which one is used during runtime as a result of their reference in code. This problem is resolved using dynamic binding in java. When type of the object is determined at run-time, it is known as dynamic binding.

Example of dynamic binding

```
class Animal{
 void eat(){
        System.out.println("animal is eating...");}
}
 class Dog extends Animal{
 void eat(){
            System.out.println("dog is eating...");}
   public static void main(String args[]){
  Animal ob=new Dog();
  ob.eat();
 }
}
```

VISIBILITY CONTROLS

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.

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. 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.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y