# CHAPTER 5

# INHERITANCE: EXTENDING A CLASS

## Inheritence:

- The mechanism of deriving a new class from an old one is called inheritance.
- The old class is known as the **base class** or **super class** or **parent class** and the new one is called the **subclass** or **derived class** or **child class**.

## Types Of Inheritance:

- Single inheritance (only one superclass)
- Multiple inheritance (several super classes)
- Hierarchical inheritance (one super class, many subclasses)
- Multilevel inheritance (Derived from a derived class)

**Note:** Java does not directly implement multiple inheritances.

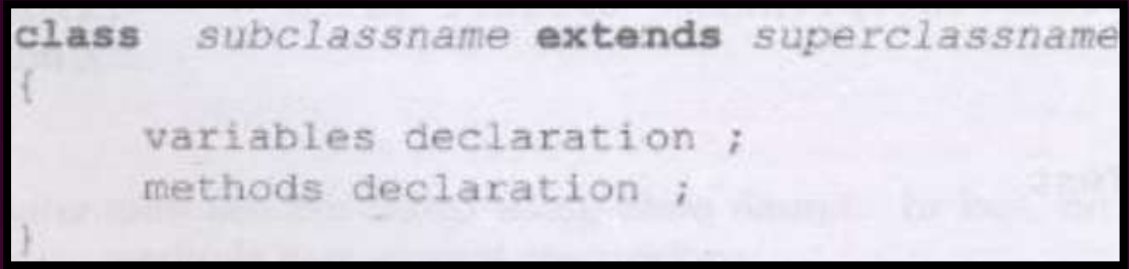
## Single Inheritance:

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

## Example:

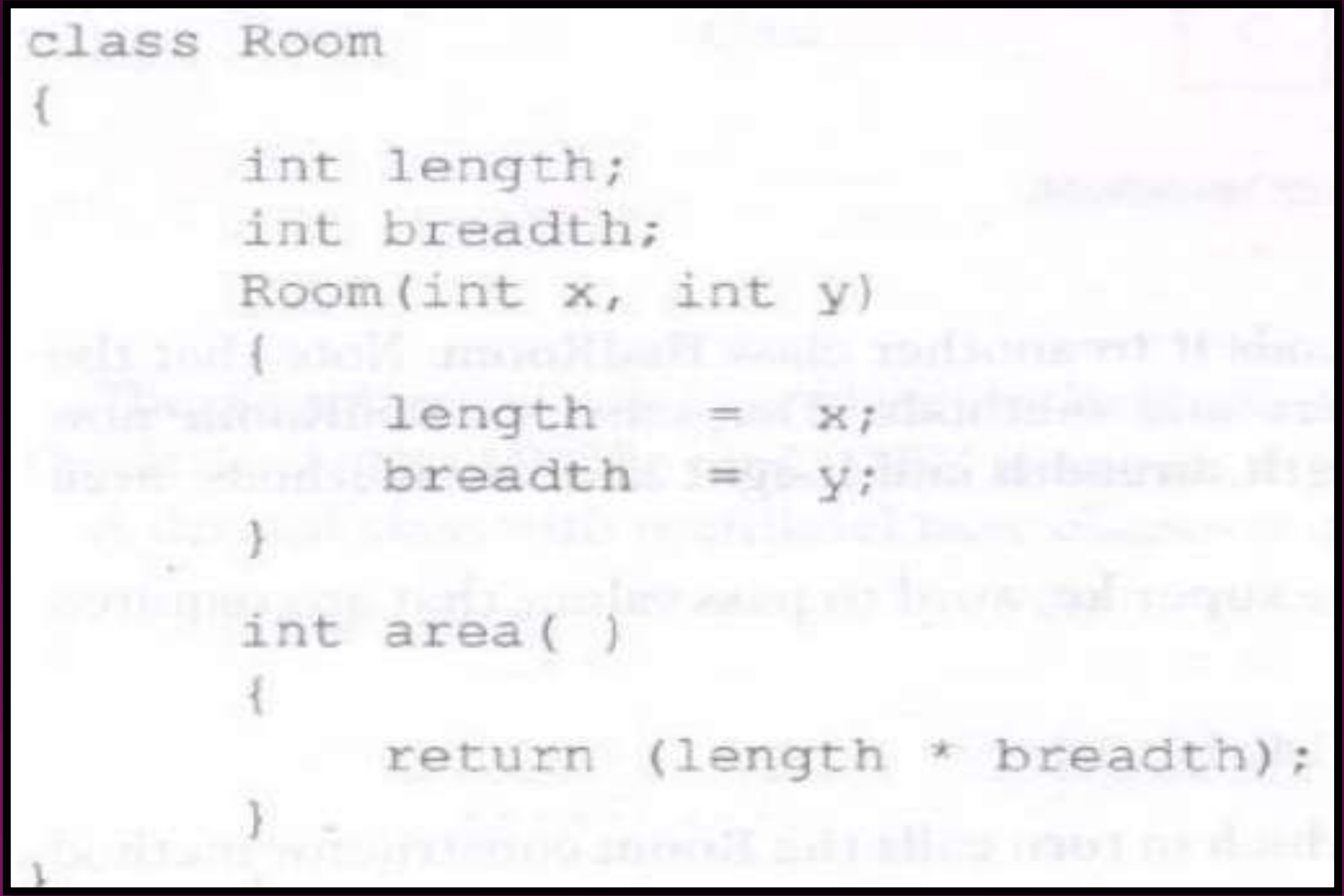
- Dog class inherits the Animal class, so there is the single inheritance.

## Defining a Subclass :



```
class  subclassname extends superclassname
{
     variables declaration ;
     methods declaration ;
}
```

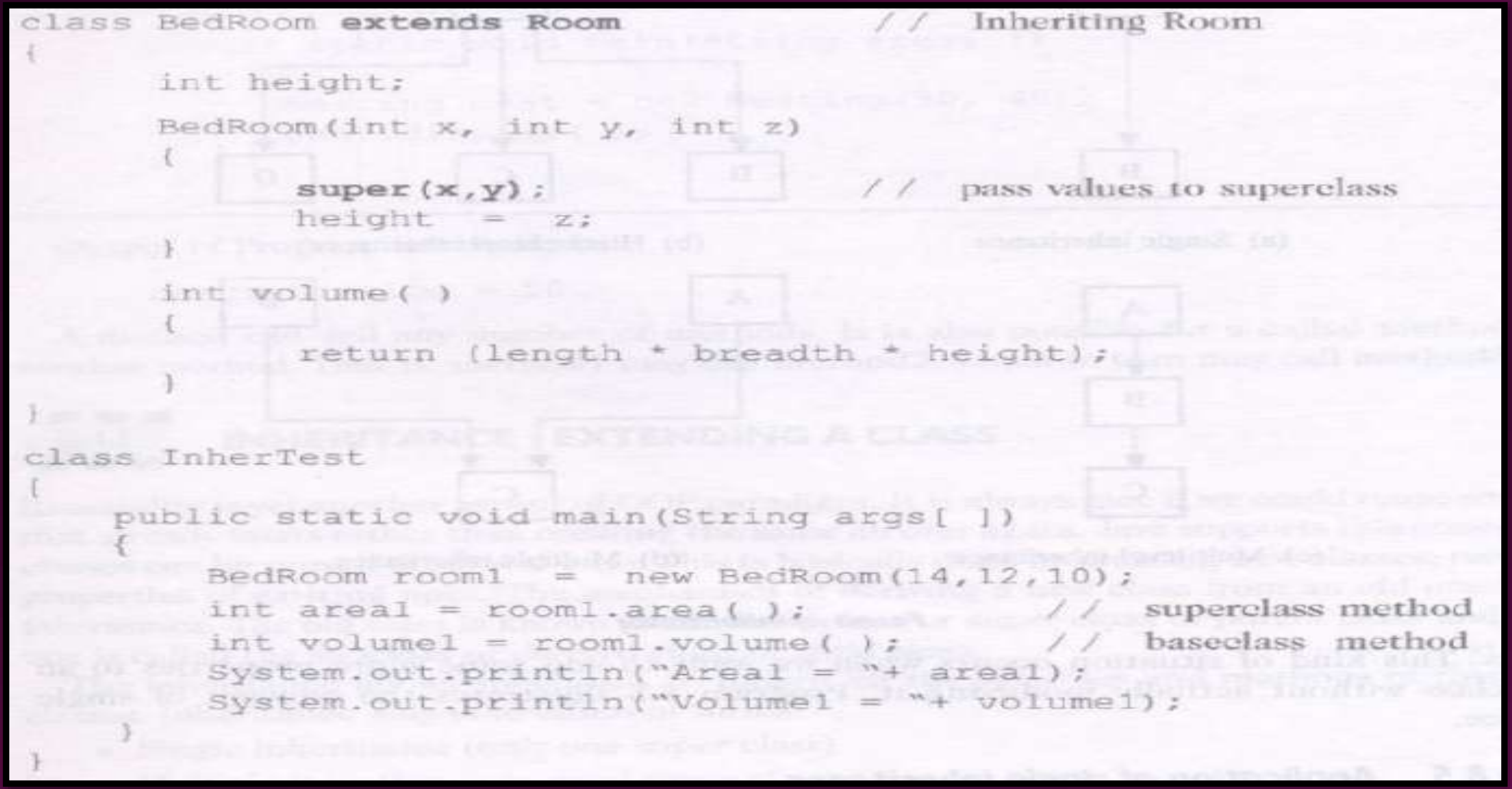
## Example:



```
class Room
{
        int length;
        int breadth;
        Room(int x, int y)
        {
            length    = x;
            breadth   = y;
        }

        int area( )
        {
            return (length * breadth);
        }
}
```

## Example Continued:



```
class BedRoom extends Room                      //  Inheriting Room
{
        int height;
        BedRoom(int x, int y, int z)
        {
                super(x,y);                     //  pass values to superclass
                height  =  z;
        }
        int volume( )
        {
                return (length * breadth * height);
        }
}
class InherTest
{
     public static void main(String args[ ])
     {
          BedRoom room1   =   new BedRoom(14,12,10);
          int area1 = room1.area( );                //  superclass method
          int volume1 = room1.volume( );            //  baseclass  method
          System.out.println("Area1 = "+ area1);
          System.out.println("Volume1 = "+ volume1);
     }
}
```

The constructor in the derived class uses the super keyword to pass values that are required by the base constructor.

The statement :BedRoom room1 = new BedRoom (14, 12, 10);

## Subclass Constructor :

- A subclass constructor is used to construct the instance variables of both the subclass and the superclass.
- The subclass constructor uses the keyword super to invoke the constructor method of the superclass.

**The keyword super is used subject to the following conditions.**

- super may only be used within a subclass constructor method
- The call to superclass constructor must appear as the first statement within the subclass constructor
- The parameters in the super call must match the order and type of the instance variable declared in the superclass

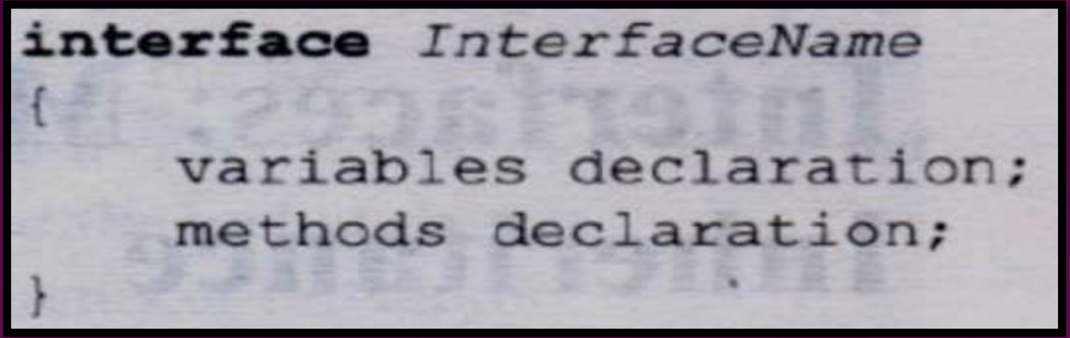
## Multiple Inheritance :

- Java does not support multiple inheritance. That is, classes in Java cannot have more than one superclass.
- For Instance, definition like

  class A extends B extends C

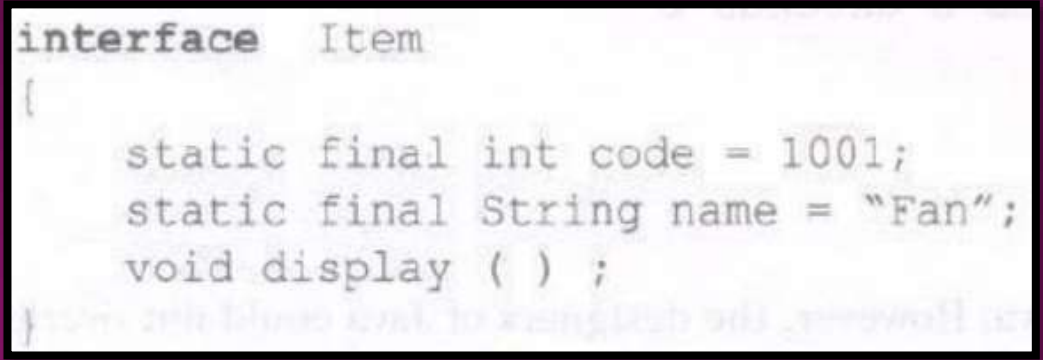
  { ---------- }
- A large number of real-life applications require the use of multiple inheritance whereby we inherit methods and properties from several distinct classes.
- Java provides an alternate approach known as interfaces to support the concept of multiple inheritance

## The general form of an interface definition is:



```
interface InterfaceName
{
      variables declaration;
      methods declaration;
}
```

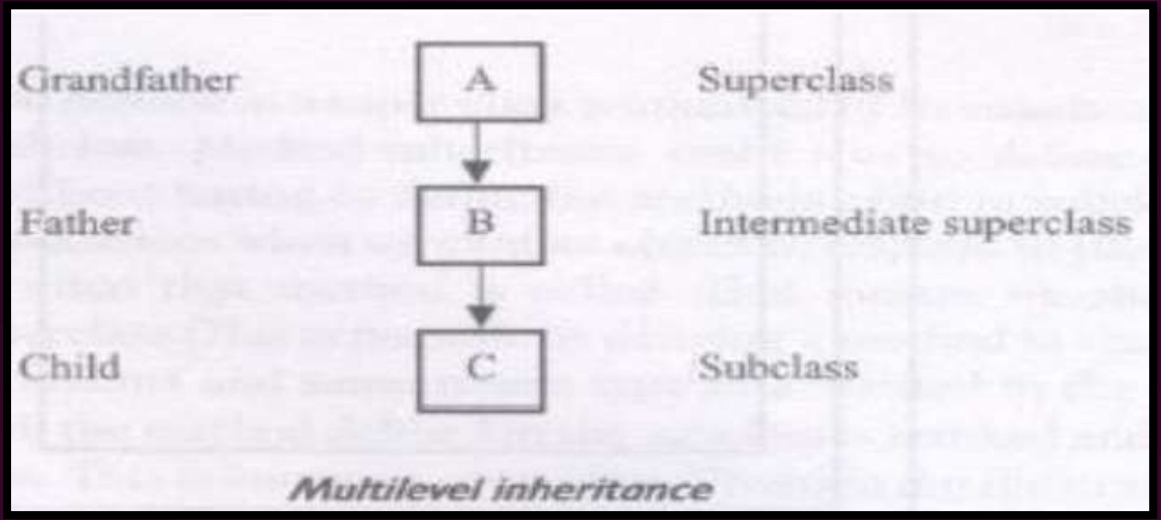
## Example:



```
interface Item
{
     static final int code = 1001;
     static final String name = "Fan";
     void display ( ) ;
}
```

## Multilevel Inheritance

- A common requirement in object-oriented programming is the use of a derived class as a super class. Java supports this concept and uses it extensively in building its class library. This concept allows us to build a chain of classes.



*Multilevel inheritance*

The class A serves as a base class for the derived class B which in turn serves as a base class for the derived class C. The chain ABC is known as **inheritance path**.

**A derived class with multilevel base classes is declared as follows.**

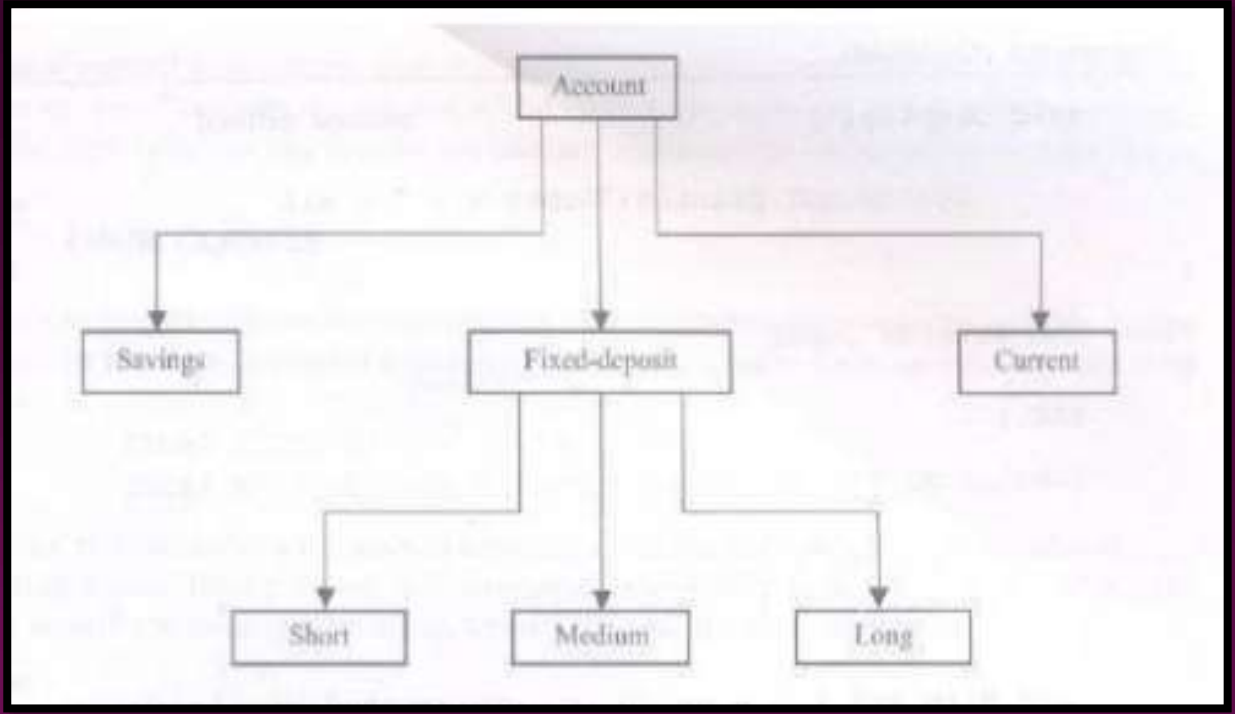
```
class A
{ …………………… …………………… }
 Class B extendsA / / Firstlevel
{ ………….. . . . . …... …… .. . ……. .......... }
Class C extends B / / Second level
{ ……………………….. ……………………….. }
```

## Hierarchical Inheritance:

- When two or more classes inherits a single class, it is known as ***hierarchical inheritance***.
- Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below the level.

**Example:**

Hierarchical classification of accounts in a commercial bank.



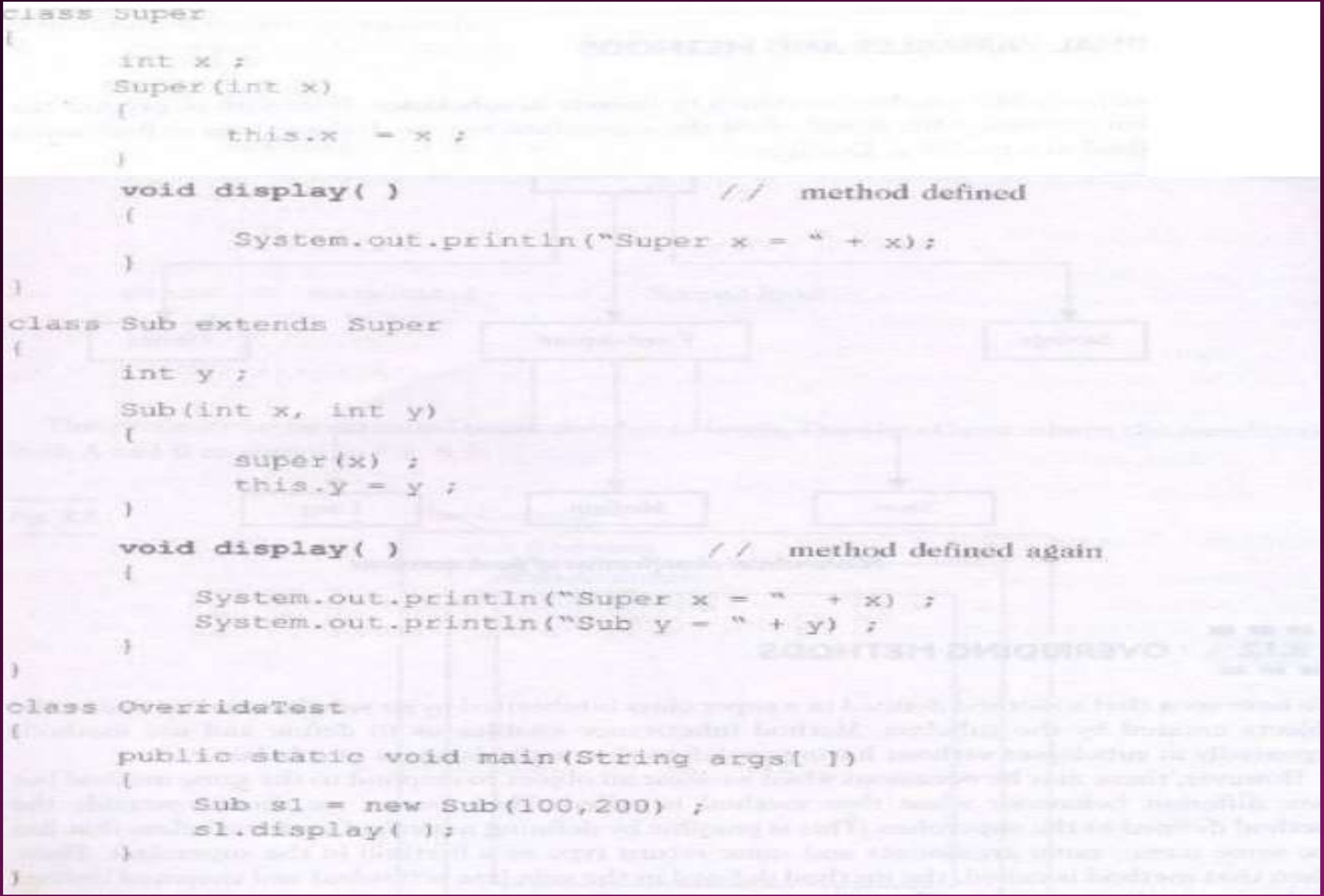
## Overloading Methods:

- In java it is possible to create methods that have the same name, but different parameter lists and different definitions. This is called method overloading.
- Method overloading is used when objects are required to perform similar tasks but using different input parameters.

## Method Overriding:

- If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.
- Method overriding is used for runtime polymorphism
- The method must have the same name as in the parent class
- The method must have the same parameter as in the parent class.

## Example:



```
class Super
{
        int x ;
        Super(int x)
        {
                this.x  = x ;
        }
        void display( )                             //   method defined
        {
                System.out.println("Super x = " + x);
        }
}
class Sub extends Super
{
        int y ;
        Sub(int x, int y)
        {
                super(x) ;
                this.y = y ;
        }
        void display( )                            //   method defined again
        {
              System.out.println("Super x = "  + x) ;
              System.out.println("Sub y = " + y) ;
        }
}
class OverrideTest
{
        public static void main(String args[ ])
        {
              Sub s1 = new Sub(100,200) ;
              s1.display( );
        }
}
```

## **FINAL VARIABLES AND METHODS :**

All methods and variables can be overridden by default in subclasses. If we wish to prevent the subclasses from overriding the members of the superclass, we can declare them as final using the keyword final as a modifier.

**Example:**

```
    final int SIZE = 100;
    final void showstatus ()
    {................ }
```

## **FINAL  CLASSES :**

A class that cannot be sub classed is called a final class.

**This is achieved using the keyword final:**

```
final class Aclass {…}
final class Bclass extends Someclass{................}
```

## Finalizer Methods:

- Java supports a concept called finalization, which is just opposite to initialization.
- Java run-time is an automatic garbage collecting system. It automatically frees up the memory resources used by the objects. But objects may hold other non-object resources such as file descriptors or window system fonts. The garbage collector cannot free these resources.
- In order to free these resources we must use a finalizer method. This is similar to destructors in C++.
- The finalizer method is simply finalize ( ) and can be added to any class.
- Java calls that method whenever it is about to reclaim the space for that object~ The finalize method should explicitly define the tasks to 'be performed.

**Abstract Methods And Classes:**

We can indicate that a method must always be redefined in a subclass, 'thus making overriding compulsory. This is done using the modifier keyword **abstract** in the method definition.

**Example:**

```
abstract class Shape
{ ABSTRACT METHODS AND CLASSES
   …..……. ..……….
   abstract void draw( );
   … …. ….. }
```

When a class contains one or more abstract methods, it should also be declared.

**While using abstract classes, we must satisfy the following conditions:**

- We cannot use abstract classes to instantiate objects directly.

**For Example:**

Shape s = new Shape ( ) is illegal because shape is an abstract class.

The abstract methods of an abstract class must be defined in its subclass.

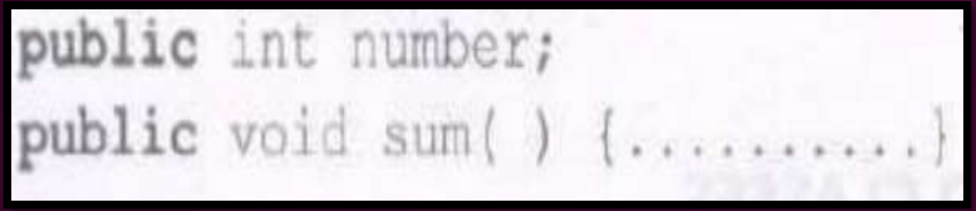
## VISIBILITY CONTROL:

- Visibility control is used to restrict the access to certain variables and methods from outside the class.
- Java provides three types of visibility modifiers: **public, private and protected.**

**public Access:**

public Visibility control is visible to all the classes even outside the class where it is defined.

**Example:**



```
public int number;
public void sum( ) {..........}
```

**friendly Access :**

When no access modifier is specified, the member defaults to a limited version of public accessibility known as "friendly" level of access.

**protected Access :**

The visibility level of a **"protected"** field lies in between the **public access and friendly access**. That is, the protected modifier makes the fields visible not only to all classes and subclasses in the same package but also to subclasses in other packages..

**private Access**

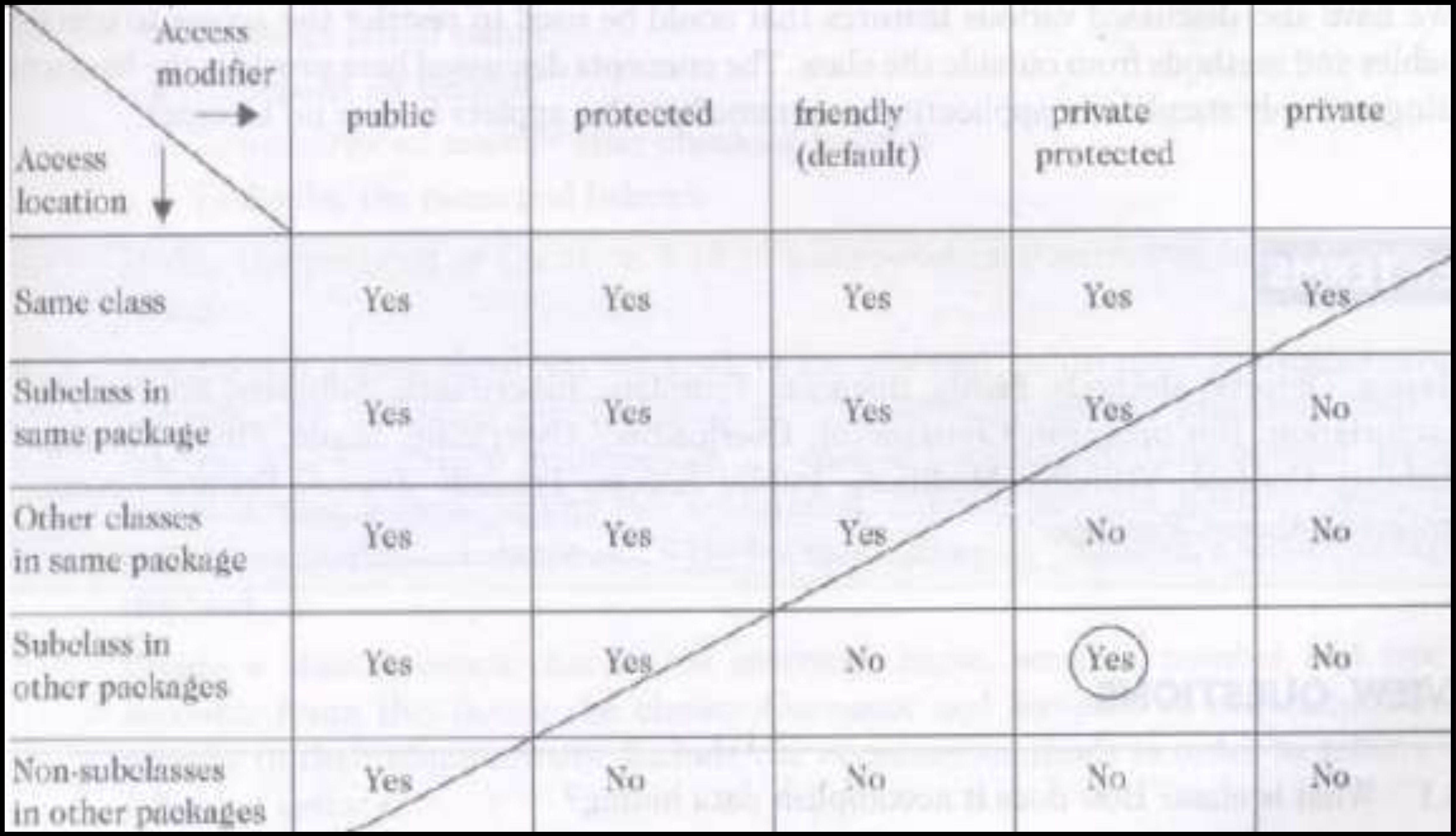
private fields enjoy the highest degree of protection. They are.

**private protected Access accessible only within their own class**

A field can be declared with two keywords private and protected together like:

**Private protected intcodeNumber;**

This gives a visibility level in between the "protected" access and "private" access. This modifier makes the fields visible in all subclasses regardless of what package they are in .



| Access modifier → / Access location ↓ | public | protected | friendly (default) | private protected | private |
|---|---|---|---|---|---|
| Same class | Yes | Yes | Yes | Yes | Yes |
| Subclass in same package | Yes | Yes | Yes | Yes | No |
| Other classes in same package | Yes | Yes | Yes | No | No |
| Subclass in other packages | Yes | Yes | No | Yes | No |
| Non-subclasses in other packages | Yes | No | No | No | No |

## Rules of Thumb:

1. Use **public** if the field is to be visible everywhere.
2. Use **protected** if the field is to be visible everywhere in the current package and also subclasses in other packages.
3. Use **"default"** if the field is to be visible everywhere in the current package only.
4. Use **private protected** if the field is to be visible only in subclasses, regardless of packages.
5. Use **private** if the field is not to be visible anywhere except in its own class.