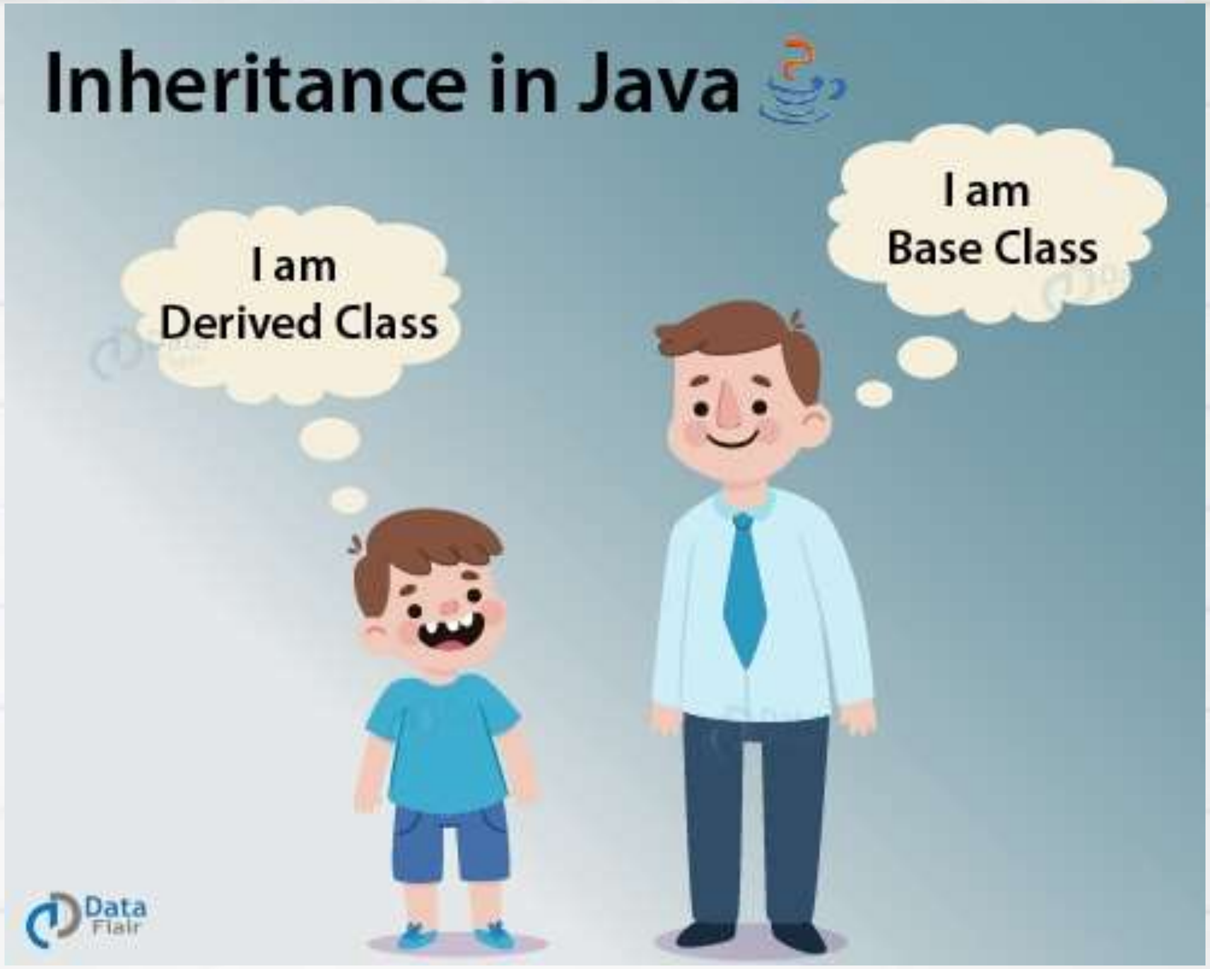
Unit 2 :-- Points to be covered

- **Inheritance**
  - Types of Inheritance (Single, Multilevel, Hierarchical)
  - Constructor in inheritance (super and this keyword)
  - Super class & subclass
  - Abstract method and classes
  - Method overriding
  - final keyword
  - super keyword
  - Implementing interfaces
  - User defined interfaces
- **Packages & Access Specifier**
  - Importing classes
  - User defined packages
  - Modifiers & Access control (Default, public, private, protected, private protected)
- **Understanding commonly used classes of java.lang package.**
  - Object class & String class
  - Wrapper classes

Inheritance



Unit 2

Introduction to Inheritance

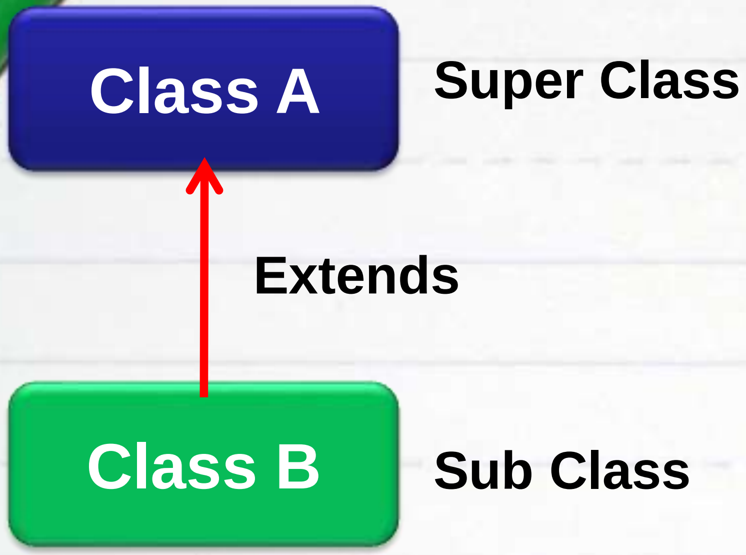
- Mechanism in Java by which one class is allowed to inherit the features(fields and methods) of another class.
- You can create new classes that are built upon existing classes.
- When you inherit from an existing class, you can reuse methods and attributes of parent class, and you can add new methods and attributes also.
- **extends** is the keyword used to inherit the properties of a class.
- Mechanism in which one object acquires all the properties and behaviors of parent object.
- Inheritance represents the parent-child relationship.

Use of Inheritance

- **For Method Overriding** - runtime polymorphism
- **For Code Reusability**.

- When a class extends another class it inherits all non-private members including attributes and methods.
- Here parent class is also known as **Super class** and child class is known as **Sub class**.

Example & Syntax



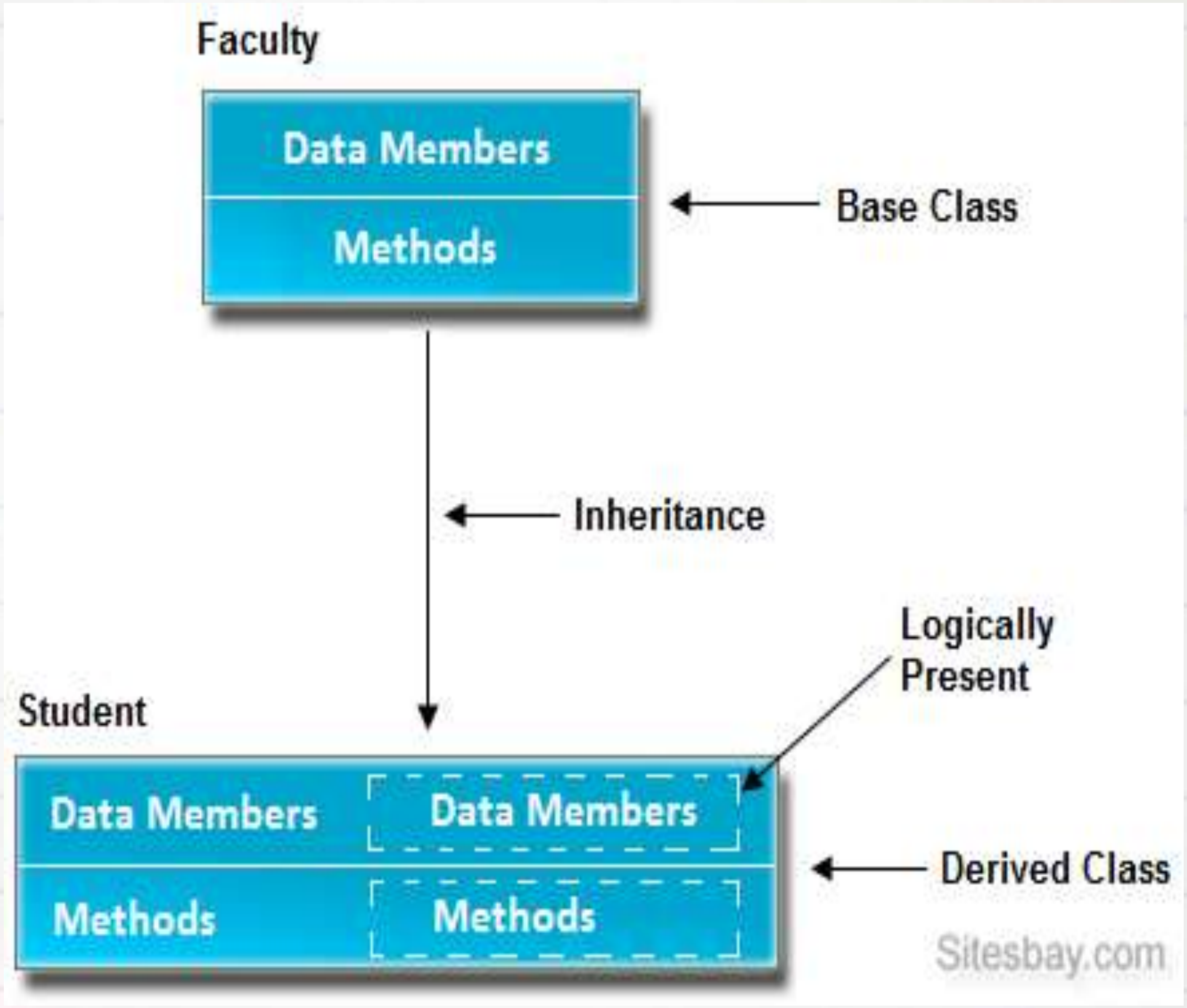
Example:

```
class A
{
        //methods and attributes
 }
class B extends A
{
        //methods and attributes
}
```

Syntax:

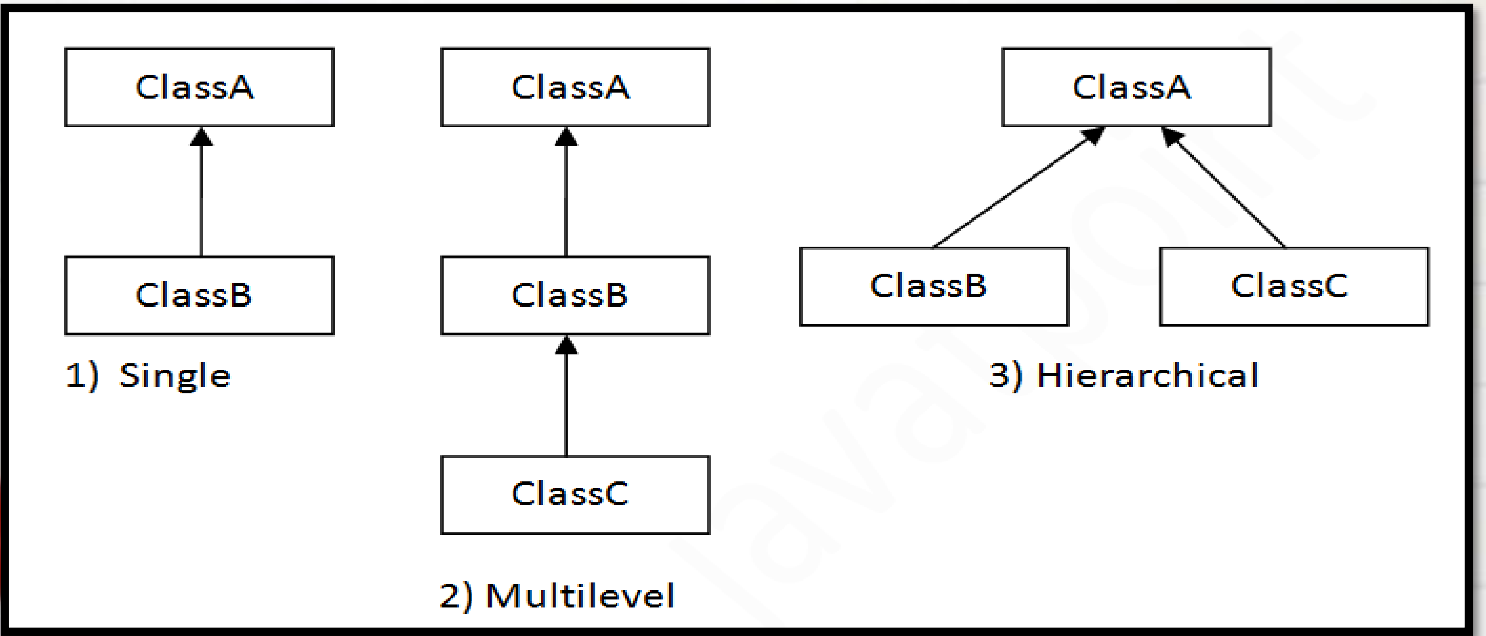
```
class Subclass-name extends Superclass-name
{
        //methods and fields
}
```

Inheritance

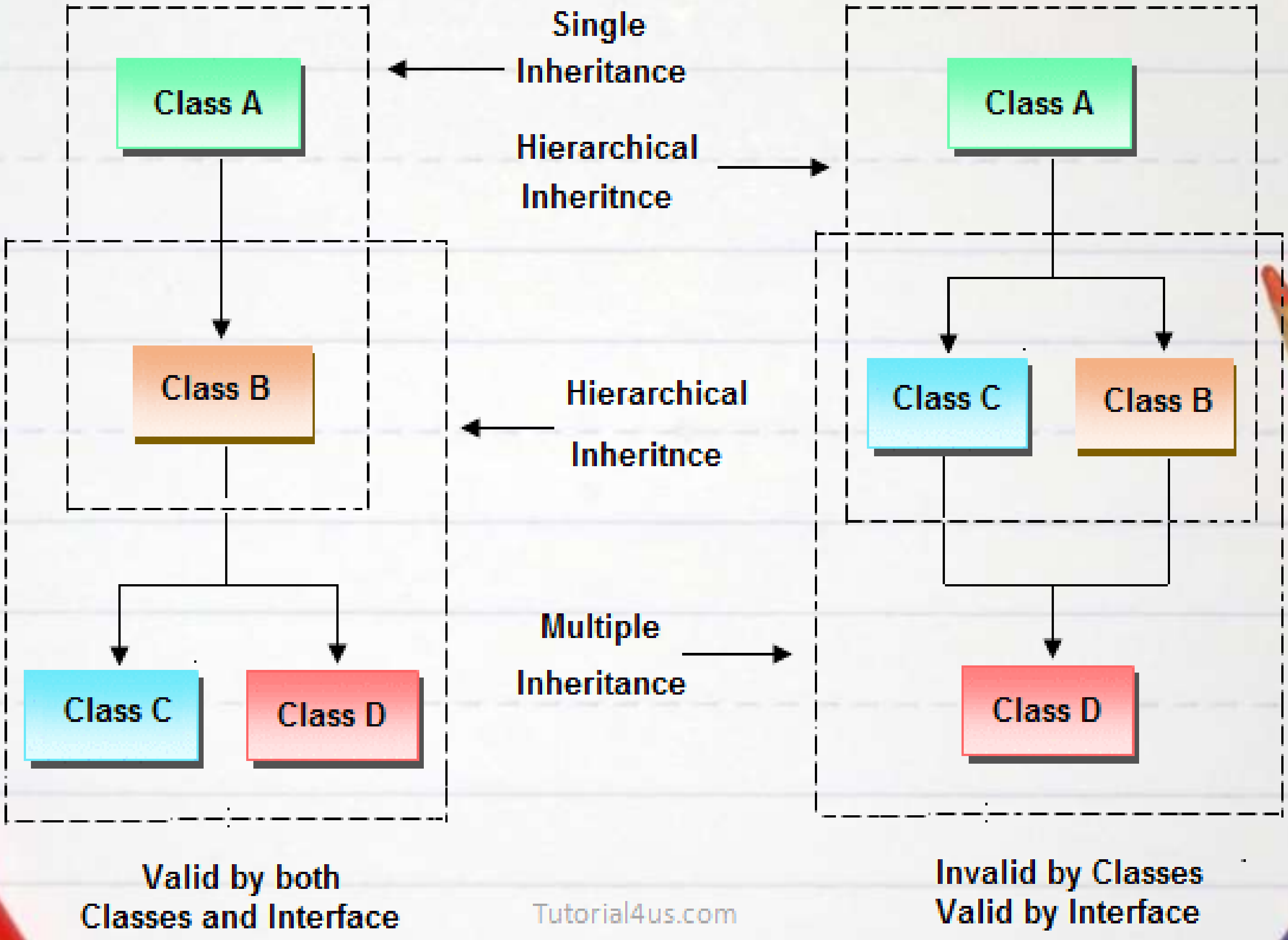


Types of Inheritance

- There can be three types of inheritance in Java:
  - Single
  - Multilevel
  - Hierarchical.
- In Java programming, **multiple and hybrid** inheritance is **not supported** through class.

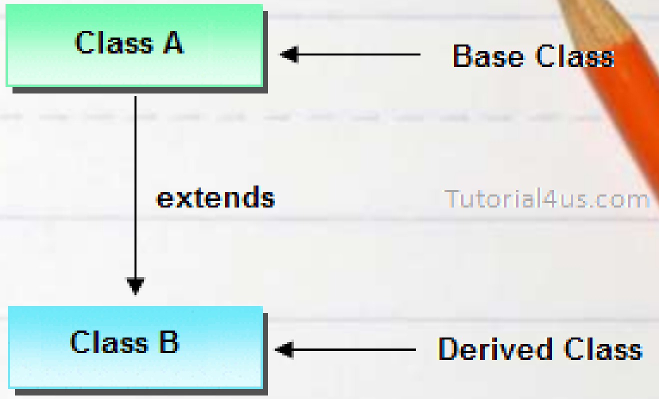


Types of Inheritance



Single Inheritance

```
class A
{    public void methodA()
     {
               System.out.println("Base class method");
     }
}
class B extends A
{    public void methodB()
     {
              System.out.println("Child class method");
     }
}
class Demo
{
     public static void main(String args[])
     {
              B obj = new B();
              obj.methodA(); //calling superclass
              obj.methodB(); //calling local method
     }
}
```



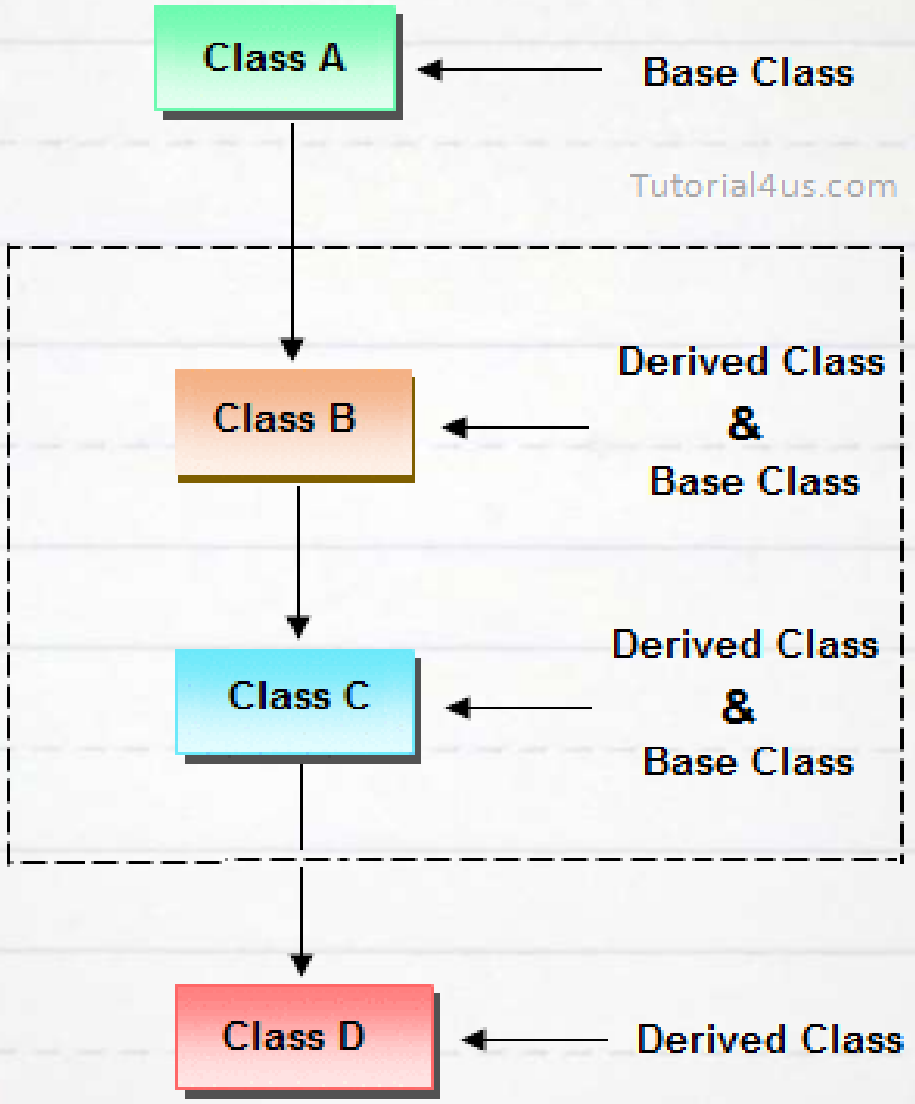
Use of super keyword

```
class A
{
      int a = 10;
 }
class B extends A
{
      int a =20;
      public void display()
      {
                 System.out.println(“value:”+a);
      }
}
class Demo
{
      public static void main(String args[])
      {
                B obj = new B();
                obj.display();     //prints value of local a variable
      }
}
```

Use of super keyword

```
class A
{
      int a = 10;
}
class B extends A
{
      int a =20;
      public void display()
      {
                System.out.println(“value:”+a);
                 System.out.println(“value:”+super.a);
      }
}
class Demo
{     public static void main(String args[])
      {
                B obj = new B();
                obj.display();
      }
 }
```

Multilevel Inheritance



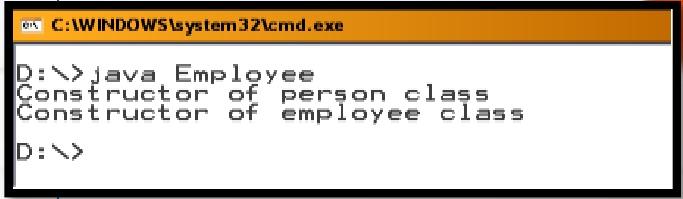
Multilevel Inheritance

```
class X
{
       public void methodX()
       {
                    System.out.println("Class X method");
       }
}
class Y extends X
{
       public void methodY()
        {
                   System.out.println("class Y method");
       }
}
class Z extends Y
{
       public void methodZ()
       {
                    System.out.println("class Z method");
       }
}
```

```
class MultiDemo
{
       public static void main(String args[])
       {
                 Z obj = new Z();
                 obj.methodX();
                 obj.methodY();
                 obj.methodZ();
       }
}
```

Constructor in Inheritance

```
class Person
{
        Person()
       {
                    System.out.println("Constructor of person class");
       }
}
class Employee extends Person
{
       Employee()
        {
                    System.out.println("Constructor of employee class");
       }
}
class ConDemo
{
        public static void main(String args[])
        {
                  Employee e1 = new Employee();
        }
}
```



```
D:\>java Employee
Constructor of person class
Constructor of employee class

D:\>
```

Note that the super class person constructor executes before the subclass Employee

Use of Super keyword in inheritance

```
class Person
{
        String fname, lname;
        Person(String fname, String lname)
        {
                    this.fname = fname;
                    this.lname= lname;
        }
}
class Student extends Person
{
        int rollno;
        String stream;
        int sem;
Student(String fname, String lname,int rollno, String stream, int sem )
{
                super(fname,lname);
                 this.rollno=rollno;
                 this.stream= stream;
                 this.sem= sem;
 }
 void display()
 {
                System.out.println(“ Name: "+fname+"    "+lname);
                System.out.println(“ Roll no: "+rollno);
                System.out.println(“ Division: " +stream+"  Sem  "+sem);
}
}
```

```
class StudDemo
{
  public static void main(String args[])
  {
        Student s1 = new  Student(“Gopi",”Rangani",30,“M.B.B.S.",2);
        s1.display();
  }
}
```

Method Overriding

- If **subclass (child class)** has the same method as declared in the **parent class**, it is known as **method overriding** in Java.
- **Same name and Same Signature but in different class having parent child relationship.**
- Method overriding is used to achieve **runtime(dynamic) polymorphism.**

Example…

```
class Bank
{
        int getInterest()
        {
                return 0;
        }
 }
class SBI extends Bank
{
          int getInterest()
         {
                return 8;
        }
}
class ICICI extends Bank
{
          int getInterest()
        {
                return 7;
        }
 }
class AXIS extends Bank
{
          int getInterest()
         {
                return 9;
         }
 }
```

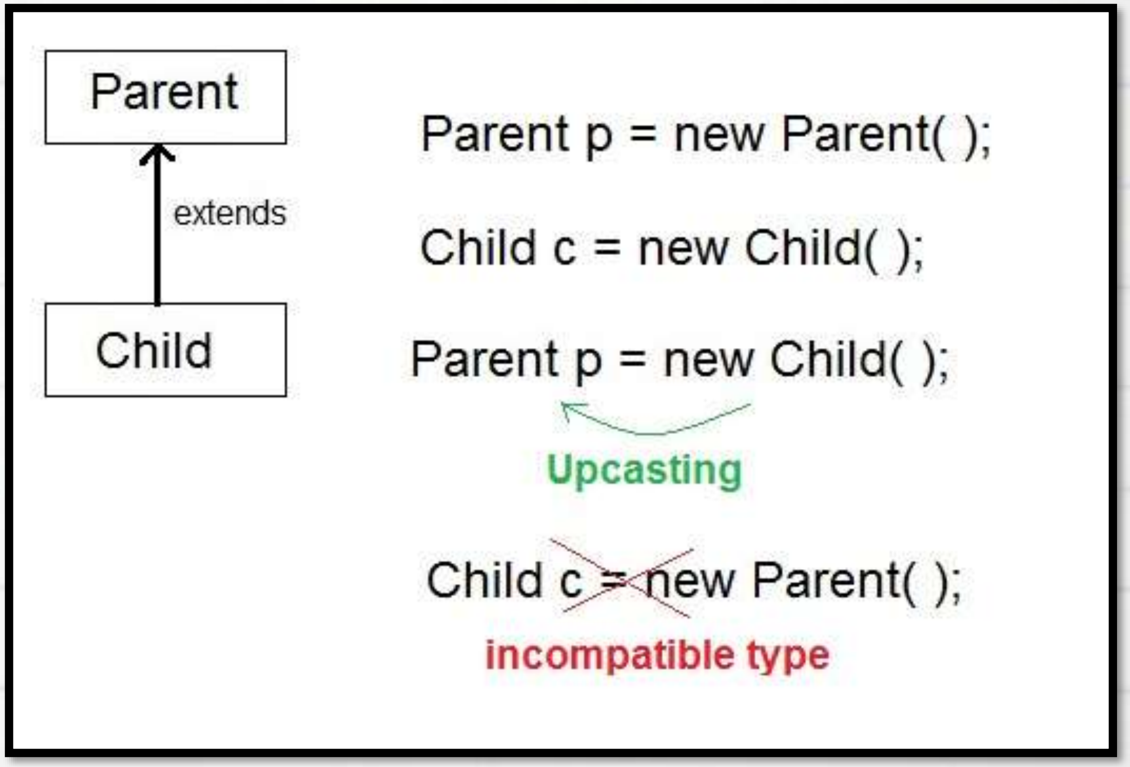
```
class Test_Bank

{

        public static void main(String args[])

        {

                SBI s=new SBI();

                ICICI i=new ICICI();

                AXIS a=new AXIS();

                System.out.println("SBI Interest Rate:"+s.getInterest());

                System.out.println("ICICI Interest Rate :"+i.getInterest());

                 System.out.println("AXIS  Interest Rate:"+a.getInterest());

        }

}
```

Method Overloading Vs. Method Overriding

Method Overloading	Method Overriding
Used to increase the readability of the program.	Used to provide the specific implementation of the method that is already provided by its super class.
Performed within a class.	Occurs in two classes that have IS-A relationship.
Parameter must be different.	Parameter must be same.
Compile time (Static) polymorphism	Run time (dynamic) polymorphism

Dynamic Method Dispatch
Dynamic Binding

- Dynamic method dispatch is the mechanism by you can achieve **Run-time Polymorphism**
- **Object of child class is handled by reference of parent class– Dynamic method dispatch**



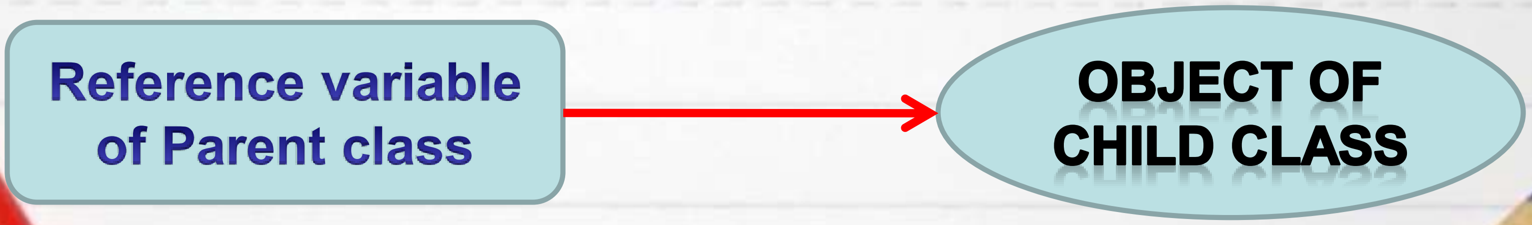
```
Parent p = new Parent( );

Child c = new Child( );

Parent p = new Child( );

Child c = new Parent( );
```

Dynamic Method Dispatch : Up casting

- When reference variable of Parent class refers to the object of Child class, it is known as upcasting.

For example:

```
class A
{
}
class B extends A
{
}
A a=new B();//upcasting
```



Dynamic Method Dispatch : Upcasting

```
class Game
 {
        void type()
        {
                      System.out.println("Indoor & outdoor");
         }
}
class Cricket extends Game
{
        void type()
        {
                     System.out.println("outdoor game");
        }
}
class Demo1
{
        public static void main(String[] args)
        {
                      Game gm = new Game();
                     Cricket ck = new Cricket();
                       gm.type();
                       ck.type();
                       gm=ck;       //gm refers to Cricket object
                       gm.type(); //calls Cricket's type
         }
 }
```

Abstract Methods and Classes

- An abstract class is a class that is declared **abstract**—it may or may not include abstract methods.
- Object of an Abstract classes cannot be created, but it can be sub classed.
- An abstract method is a method that is declared without an implementation
- **abstract Returntype methodName(argu list);**
- **If any class includes abstract methods, the class itself must be declared as abstract.**
- If you are extending any abstract class having abstract method, you must either provide the implementation of the method or make this sub class also abstract.

Example of abstract keyword

```
abstract class Shape
{
        abstract void draw();
}
class Rectangle extends Shape
{
       void draw()
        {
            System.out.println("drawing rectangle");
       }
}
class Circle extends Shape
{
       void draw()
       {
                  System.out.println("drawing circle");
       }
}
```

```
class Test_Abstract
{
        public static void main(String args[])
       {
           Shape s;
           s=new Circle();
           s.draw();
           s= new Rectangle();
           s.draw();
       }
 }
```

An abstract class must have atleast one abstract method….

```
abstract class Bike
{
        Bike()
       {
                System.out.println(“Bike is created");
       }
       abstract void run();
       void changeGear()
       {
                System.out.println("gear changed");
       }
 }
class Honda extends Bike
{
        void run()
       {
                System.out.println("running safely..");
       }
}
```

```
class Demo_Abstraction
{
         public static void main(String args[])
       {
                Bike obj = new Honda();
                obj.run();
                obj.changeGear();
       }
 }
```

Output:

```
Bike is created
running safely..
gear changed
```

Final class, variables and methods

- Java classes declared as final cannot be extended. Restricting inheritance!
- A java variable can be declared using the keyword final. Then the final variable can be assigned only once. **–Constant variable**
- Methods declared as final cannot be overridden.

final keyword

- The **final keyword** in Java is used to restrict the user.
- final can be:
  - variable
  - method
  - Class

- **Java final variable:** if you make variable as final you can not reinitialized it. Which used to declare a **constant** variable.

- **Java final method:** If you make any method as final, you cannot override it.

- **Java final class :**If you make any class as final, you cannot extend it.

Example of final method

```
class Bike
{
        final void run()
        {
                System.out.println("running");
        }
}
class Honda extends Bike
{
       void run()
       {
                System.out.println("running safely with 100kmph");
       }
}
class FinalDemo
{
   public static void main(String args[])
  {
        Honda honda= new Honda();
        honda.run();
  }
}
```

Output:
Compile time error

Example of final class

```
final class Bike
{
        void run()
       {
                  System.out.println("running ");
        }
}
class Honda extends Bike ✗
{
         void run()
         {
                  System.out.println("running safely ");
         }
}
class FinalDemo
{
         public static void main(String args[])
       {
                   Honda1 honda= new Honda();
                    honda.run();
       }
}
```

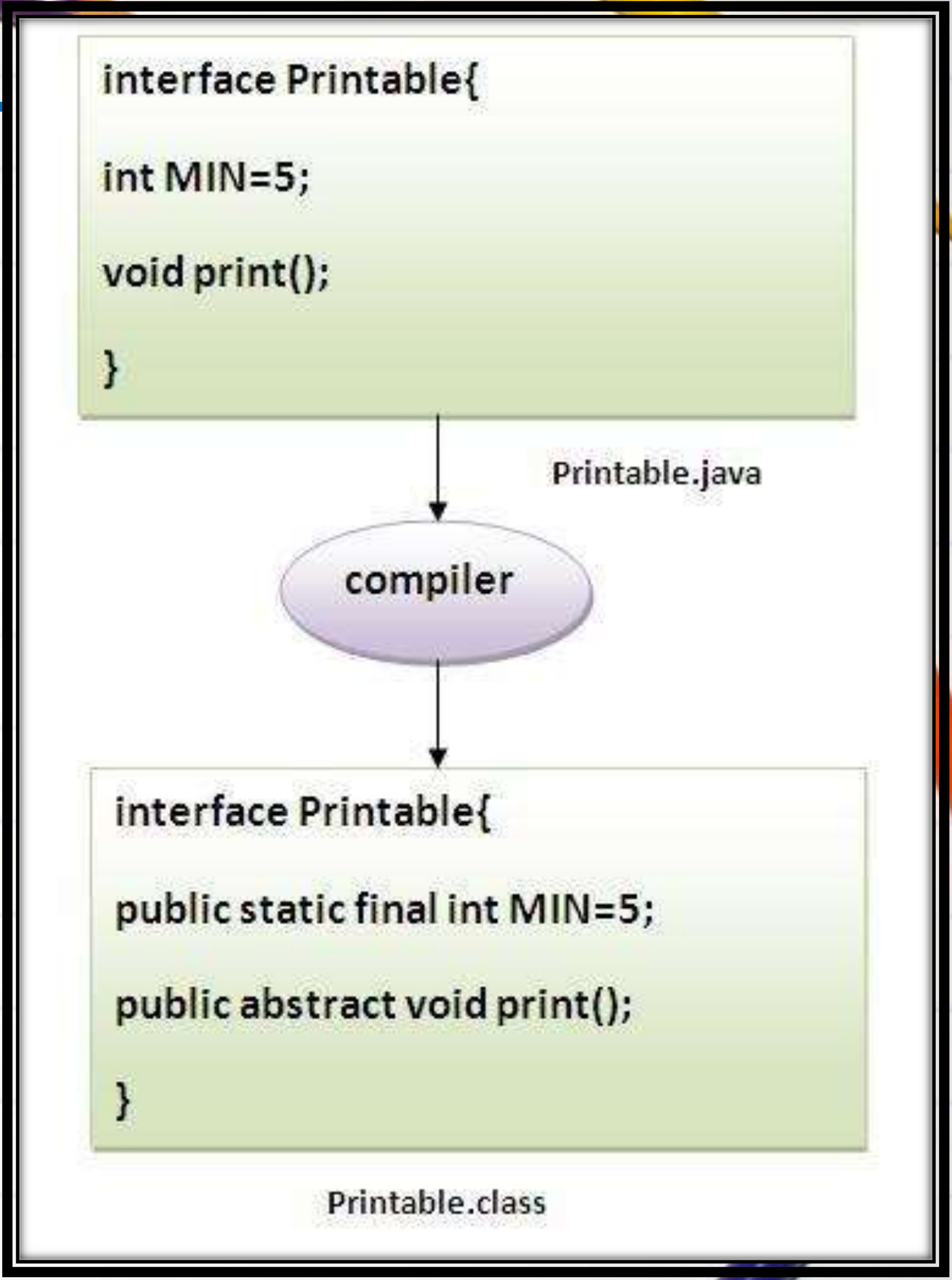
Output:
Compile time error

Introduction to Interface

- An interface in Java is a blueprint of a class.
- It has static constants and abstract methods only.
- There can be only abstract methods in the Java interface not method body.
- It is used to achieve **fully abstraction** and **multiple inheritance** in Java.
- Java Interface also **represents IS-A relationship**.
- It cannot be instantiated just like abstract class.

Interface…

- **Note:**
- The java compiler adds **public** and **abstract** keywords before the interface method
- **public, static** and **final** keywords before data members



```
interface Printable{
int MIN=5;
void print();
}
```

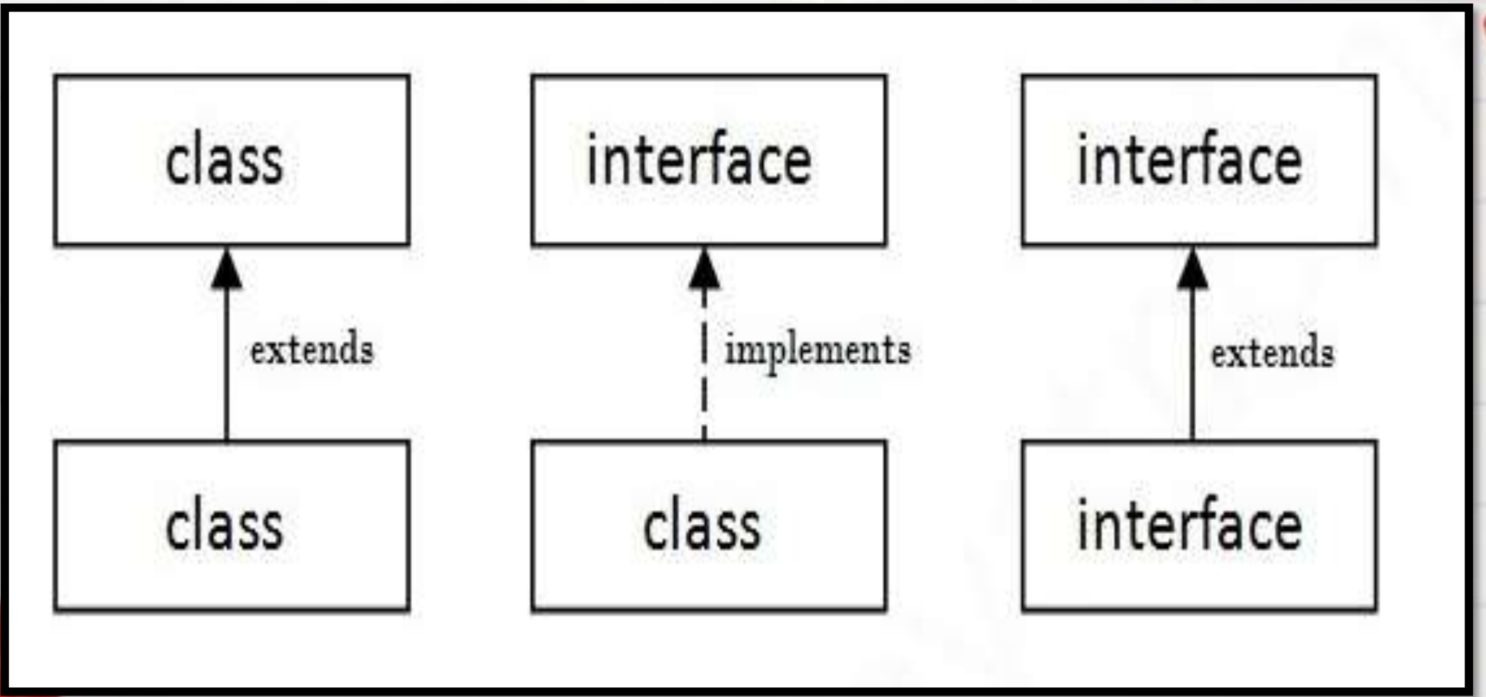
```
interface Printable{
public static final int MIN=5;
public abstract void print();
}
```

Interface…

- You cannot instantiate an interface. (Can’t create an object)
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

Class & Interface…

- A class extends another class
- An interface extends another interface
- **class implements an interface**.



Declaring Interfaces

- The interface keyword is used to declare an interface.
- Syntax:

```
interface NameOfInterface
{
        //Any number of final, static fields
        //Any number of abstract method declarations
 }
```

Example

```
interface printable
{
        void print();
}
class Test_Interface implements printable
{
        public void print()
        {
                System.out.println("Hello");
        }
}
class IntDemo
{
        public static void main(String args[])
        {
                Test_Interface obj = new Test_Interface();
                obj.print();
        }
}
```

Example…

```
interface Shape
{
    String LABLE="Shape";
    void draw();
    double getArea();
}
 class Circle implements Shape
{
     double radius;
     Circle(double r)
     {
          this.radius = r;
     }
     public void draw()
     {
            System.out.println("Drawing Circle");
      }
     public double getArea()
     {
          return 3.14*r*r;
       }
}
```

```
class ShapeTest
{
    public static void main(String[] args)
       {
                Shape s = new Circle(10);
                s.draw();
          System.out.println("Area=“ + s.getArea());
   }
 }
```

Output:

```
Drawing Circle
Area=314.1592653589793
```

Extending Interfaces:

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

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

The finalize( ) Method

- Sometimes an object will need to perform some action when it is destroyed.
- For example, if an object is holding some non-Java resource such as a file, then you might want to make sure these resources are freed before an object is destroyed.
- To handle such situations, Java provides a mechanism called finalization.
- By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.
- finalize( ) is only called just prior to garbage collection.
- The finalize( ) method has this general form:

```
protected void finalize( )
{
        // finalization code here
}
```