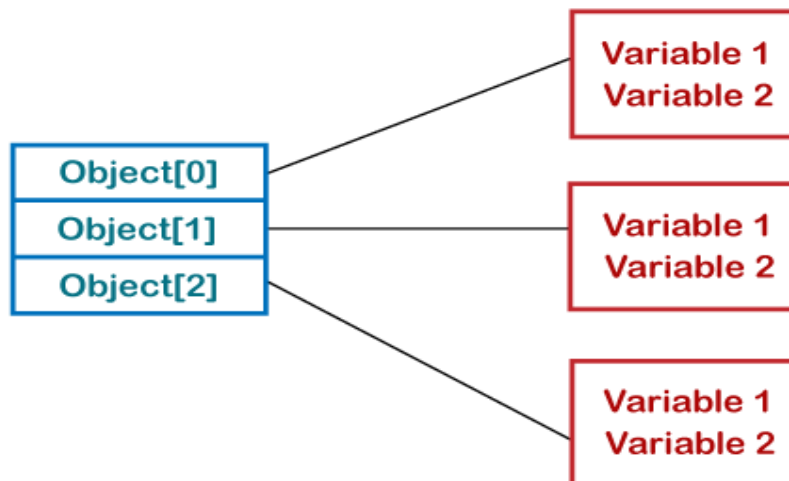


## Chapter 2 – Unit 2

### Array of Objects

- Java is an object-oriented programming language.
- Most of the work done with the help of **objects**.
- We know that an array is a collection of the same data type that dynamically creates objects and can have elements of primitive types.
- Java allows us to store objects in an array.
- In Java, the class is also a user-defined data type.
- An array that contains class type elements are known as an array of objects.

#### Arrays of Objects



- **Note:**In Java, we can create arrays by using new operator and we know that every object is created using new operator. Hence we can say that array is also an object.

## Creating an Array of Objects

- Before creating an array of objects, we must create an instance of the class by using the new keyword.
- We can use any of the following statements to create an array of objects.
- **Syntax:**
- `ClassName objname[]=new ClassName[array_length];` //declare and instantiate an array of objects .
- **Or**  
`ClassName[] objArray;`
- **Or**  
`ClassName objArray[];`
- Suppose, we have created a class named Employee. We want to keep records of 20 employees of a company having three departments. In this case, we will not create 20 separate variables. Instead of this, we will create an array of objects, as follows.

```
Employee department1[20];
```

```
Employee department2[20];
```

```
Employee department3[20];
```

- The above statements create an array of objects with 20 elements.
- In the following program, we have created a class named Product and initialized an array of objects using the constructor. We have created a constructor of the class Product that contains product id and product name. In the main function, we have created individual objects of the class Product. After that, we have passed initial values to each of the objects using the constructor.

```
class ArrayOfObjects
```

```
{
```

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

```
{
```

```
//create an array of product object.
```

```
Productobj[] = new Product[5] ;
```

```
//create & initialize actual product objects using constructor
```

```
obj[0] = new Product(23907,"Dell Laptop");
```

```
obj[1] = new Product(91240,"HP 630");
```

```
obj[2] = new Product(29823,"LG OLED TV");
```

```
obj[3] = new Product(11908,"MI Note Pro Max 9");
```

```
obj[4] = new Product(43590,"Kingston USB");
//display the product object data
System.out.println("Product Object 1:");
obj[0].display();
System.out.println("Product Object 2:");
obj[1].display();
System.out.println("Product Object 3:");
obj[2].display();
System.out.println("Product Object 4:");
obj[3].display();
System.out.println("Product Object 5:");
obj[4].display();
}
}
//Product class with product Id and product name as attributes
class Product
{
    int pro_Id;
    String pro_name;
    //Product class constructor
    Product(int pid, String n)
    {
```

```
pro_Id = pid;
pro_name = n;
}
void display()
{
System.out.print("Product Id = "+pro_Id + " " + " Product Name =
"+pro_name);
System.out.println(" ");
}
}
```

**Output :**

```
E:\>javac ArrayOfObjects.java
E:\>java ArrayOfObjects
Product Object 1:
Product Id = 11    Product Name = Dell Laptop
Product Object 2:
Product Id = 12    Product Name = HP 630
Product Object 3:
Product Id = 13    Product Name = LG OLED TV
Product Object 4:
Product Id = 14    Product Name = MI Note Pro Max 9
Product Object 5:
Product Id = 15    Product Name = Kingston USB
```

## Inheritance:Basic concepts

- **Inheritance is a mechanism in which one object acquires all the properties and behaviors of a parent object.**
- 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 represents the IS-A relationship which is also known as a parent-child relationship.
- **Why use inheritance:**
  - 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:** As the name specifies, 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:**

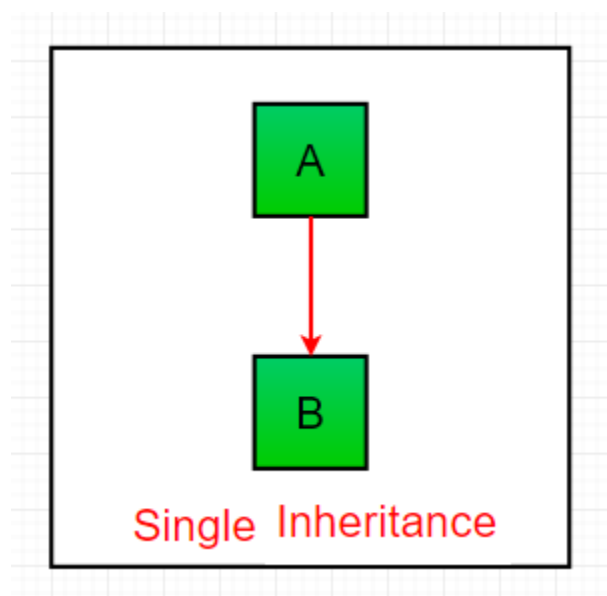
```
class Superclass-name
{
    //methods and fields
}

class 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. The meaning of "extends" is to increase the functionality.
- In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

- **Types of Inheritance in Java:**

- **1. Single Inheritance:** In single inheritance, subclasses inherit the features of one superclass. In the image below, class A serves as a base class for the derived class B.





// Java program to illustrate the concept of single inheritance.

```
class one
```

```
{
```

```
void print1()
```

```
{
```

```
System.out.println("This is Base Class method");
```

```
}
```

```
}
```

```
class two extends one
```

```
{
```

```
void print2()
```

```
{
```

```
System.out.println("This is Derived Class method");
```

```
}
```

```
}
```

```
class demo
```

```
{
```

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

```
{
```

```
one o = new one();  
o.print1();  
two t = new two();  
t.print1();  
t.print2();  
}  
}
```

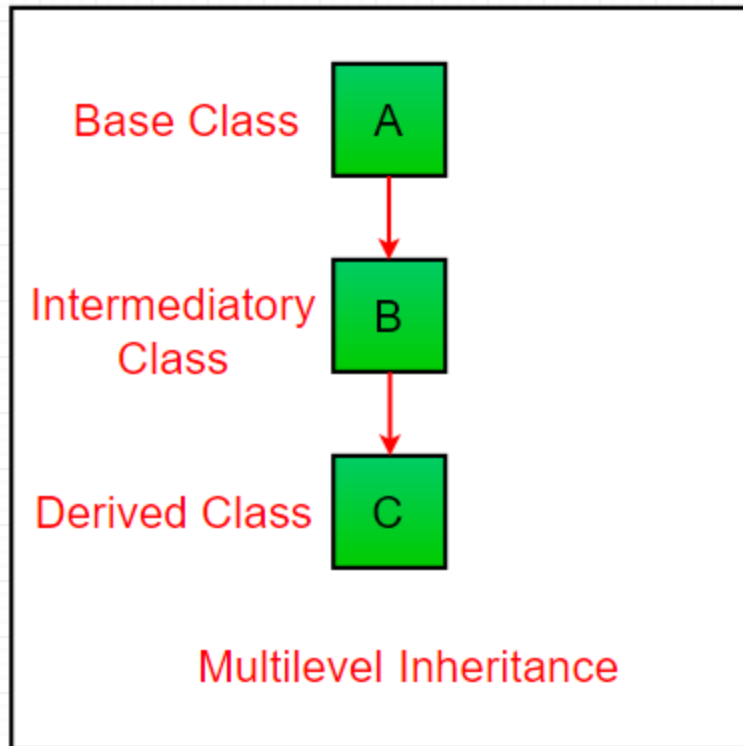
**Output :**

This is Base Class method

This is Base Class method

This is Derived Class method

- **2. Multilevel Inheritance:** In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In the below image, 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. In Java, a class cannot directly access the grandparent's members.



```
class one
```

```
{
```

```
void print1()
```

```
{
```

```
System.out.println("This is Base Class method");
```

```
}
```

```
}
```

```
class two extends one
```

```
{
```

```
void print2()
```

```
{
```

```
System.out.println("This is Intermediate Class method");
```

```
}  
  
}  
  
class three extends two  
{  
    void print3()  
    {  
        System.out.println("This is Derived Class method");  
    }  
}  
  
class demo  
{  
    public static void main(String[] args)  
    {  
        one o = new one();  
        o.print1();  
        two t = new two();  
        t.print1();  
        t.print2();  
        three th = new three();  
        th.print1();  
        th.print2();  
        th.print3();  
    }  
}
```

}

}

### Output :

This is Base Class method

This is Base Class method

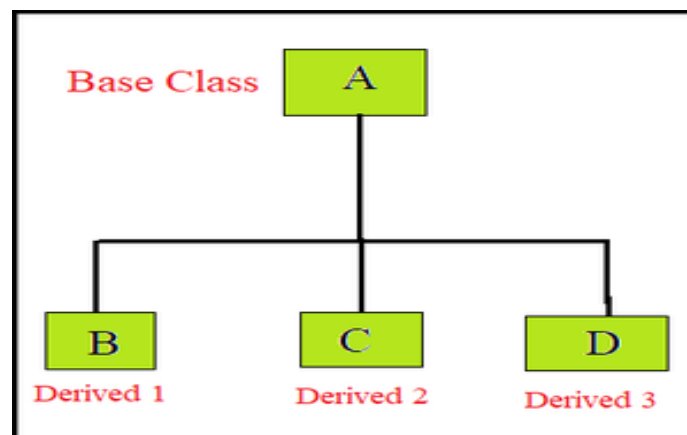
This is Intermediate Base Class method

This is Base Class method

This is Intermediate Base Class method

This is Derived Class method

- **3. Hierarchical Inheritance:** In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In the below image, class A serves as a base class for the derived class B, C and D.



```
class one
{
void print1()
{
System.out.println("This is One Class method");
}
}

class two extends one
{
void print2()
{
System.out.println("This is Two Class method");
}
}

class three extends one
{
void print3()
{
System.out.println("This is Three Class method");
}
}

class four extends one
```

```
{  
void print4()  
{  
System.out.println("This is four Class method");  
}  
}  
  
class demo  
{  
public static void main(String[] args)  
{  
one o = new one();  
o.print1();  
two t = new two();  
t.print1();  
t.print2();  
three th = new three();  
th.print1();  
th.print3();  
four f = new four();  
f.print1();  
f.print4();  
}
```

```
}
```

**Output :**

This is One Class method

This is One Class method

This is Two Class method

This is One Class method

This is Three Class method

This is One Class method

This is four Class method

**Method overriding**

- 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.
- The purpose of Method Overriding is that if the derived class wants to give its own implementation it can give by overriding the method of the parent class. When we call this overridden method, it will execute the method of the child class, not the parent class.
- Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes.



- When a method in a subclass has the same name, same parameters or signature, and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to *override* the method in the super-class.
- For **Method Overriding** (so **runtime polymorphism** can be achieved).

**Example 1:**

```
class base
```

```
{
```

```
void show()
```

```
{
```

```
System.out.println("This is Base Class Method");
```

```
}
```

```
}
```

```
class derived extends base
```

```
{
```

```
void show()
```

```
{
```

```
System.out.println("This is Derived Class Method");
```

```
}
```

```
}
```

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

**Output:**

This is Derived Class Method

**Example 2:**

```
class A
{
int i, j;
A(int a, int b)
{
i = a;
j = b;
}
// display i and j
void show()
{
```

```
System.out.println("This is Base Class Method");
System.out.println("i and j: " + i + " " + j);
}
}
class B extends A
{
int k;
B(int a, int b, int c)
{
super(a, b);
k = c;
} //display k – this overrides show() in A
void show()
{
System.out.println("This is Derived Class Method");
System.out.println("k: " + k);
}
}
class Override
{
public static void main(String args[])
{
```

```
B obj = new B(1, 2, 3);  
obj.show(); // this calls show() in B  
}  
}
```

**The output produced by this program is shown here:**

k: 3

**Note:**

- **Method overriding is one of the way by which java achieve Run Time Polymorphism , i.e, The version of a method that is executed will be determined by the object that is used to invoke it.**
- If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed.
- In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.

## Usage of super key word

- The most common use of the super keyword is to eliminate the confusion between super classes and subclasses that have methods or variables with the same name.

### Usage of super Keyword is:

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

### Example of First Point:

**Use of super with variables:** This scenario occurs when a derived class and base class has same data members. In that case there is a possibility of ambiguity for the JVM.

```
class base
{
int a = 100;
}
class derived extends base
{
int a = 200;
```

```
void display()
{
    System.out.println("Super class member: " + super.a);
    System.out.println("sub class member: " + a);
}
}

class Test
{
    public static void main(String[] args)
    {
        derived d = new derived();
        d.display();
    }
}
```

### **Output**

Super class member : 100

Subclass member : 200

**Note :**In the above example, both super class and subclass have a member 'a'. We could access 'a' of base class in subclass using super keyword.

### Example of SecondPoint:

**Use of super with methods:** This is used when we want to call parent class method. So whenever a parent and child class have same named methods then to resolve ambiguity we use super keyword. This code snippet helps to understand the said usage of super keyword.

```
class base
{
void message()
{
System.out.println("This is base class method");
}
}

class derived extends base
{
void message()
{
System.out.println("This is derived class method");
}

void display()
{
// will invoke or call current class message() method
message();
}
```

```
// will invoke or call parent class message() method
super.message();
}
}

/* Driver program to test */
class Test
{
    public static void main(String args[])
    {
        derived d = new derived();
        d.display();
    }
}
```

## Output

This is Derived class method

This is Base class method

## Example of Third Point

**Use of super with constructors:** super keyword can also be used to access the parent class constructor. One more important thing is that, "super" can call both parametric as well as non-parametric constructors depending upon the situation. Following is the code snippet to explain the above concept:



```
class base
{
    base()
    {
        System.out.println("base class Constructor");
    }
}

class derived extends base
{
    derived()
    {
        super();
        System.out.println("derived class Constructor");
    }
}

class Test
{
    public static void main(String[] args)
    {
        derived c = new derived();
    }
}
```

## **Output:**

base class Constructor

derived class Constructor

## **Other Important points:**

- Call to `super()` must be first statement in `Derived(Student)` Class constructor.
- If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the superclass does not have a no-argument constructor, you will get a compile-time error. `Object` does have such a constructor, so if `Object` is the only superclass, there is no problem.
- If a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, you might think that a whole chain of constructors called, all the way back to the constructor of `Object`. This, in fact, is the case. It is called constructor chaining..
- **Is Super mandatory in Java?**
- However, using `super()` is not compulsory. Even if `super()` is not used in the subclass constructor, the compiler implicitly calls the default constructor of the superclass.

## **Abstract class**

### **Abstract class:**

- Java Abstract class is used to provide common method implementation to all the subclasses or to provide default implementation. (or) The purpose of an abstract class is to provide a common definition of a base class that multiple derived classes can share.
- Any class that contains one or more abstract methods must also be declared abstract.
- 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.

### **Points to Remember:**

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- 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.

### **Example of abstract class:**

**abstract class classname**

```
{  
}
```

### **Abstract Method:**

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

Example of abstract method

```
abstract void methodname(); //no method body
```

**Example1 :A Simple demonstration of abstract.**

```
abstract class A
```

```
{
```

```
    abstract void callme();
```

```
    // concrete methods are still allowed in abstract classes
```

```
    void callmetoo()
```

```
{
```

```
    System.out.println("This is a concrete method.");
```

```
}
```

```
}
```

```
class B extends A
```

```
{
```

```
void callme()
{
    System.out.println("B's implementation of callme.");
}

class AbstractDemo
{
    public static void main(String args[])
    {
        //A a = new A(); A is abstract class , cannot be instantiated.
        B b = new B();
        b.callme();
        b.callmetoo();
    }
}
```

**Output:**

B's implementation of callme.

This is a concrete method.

## Example 2:

```
abstract class A
{
    abstract void callme();

    // concrete methods are still allowed in abstract classes
    void callmetoo()
    {
        System.out.println("This is a concrete method.");
    }
}

class B extends A
{
    void callme()
    {
        System.out.println("B's implementation of callme.");
    }
}

class C extends A
{

```

```
void callme()
{
    System.out.println("C's implementation of callme.");
}

}

class AbstractDemo
{
    public static void main(String args[])
    {
        //A a = new A(); A is abstract class , cannot be instantiated.
        B b = new B();
        C c = new C();
        b.callme();
        b.callmetoo();
        c.callme();
        c.callmetoo();
    }
}
```

**Output:**

B's implementation of callme.

This is a concrete method.

C's implementation of callme.

This is a concrete method.

**Example 3: // Using abstract methods and classes.**

```
abstract class Figure
{
    double dim1;
    double dim2;
    Figure(double a, double b)
    {
        dim1 = a;
        dim2 = b;
    }
    // area is now an abstract method
    abstract double area();
}
```



```
class Rectangle extends Figure
{
    Rectangle(double a, double b)
    {
        super(a, b);
    }

    // override area for rectangle
    double area()
    {
        System.out.println("Inside Area for Rectangle.");
        return dim1 * dim2;
    }
}

class Triangle extends Figure
{
    Triangle(double a, double b)
    {
        super(a, b);
    }
}
```

```
// override area for right triangle

double area() {

System.out.println("Inside Area for Triangle.");

return dim1 * dim2 / 2;

}

}

class AbstractAreas

{

public static void main(String args[])

{

// Figure f = new Figure(10, 10); // illegal now

Rectangle r = new Rectangle(9, 5);

Triangle t = new Triangle(10, 8);

Figure figref; // this is OK, no object is created

figref = r;

System.out.println("Area is " + figref.area());

figref = t;

System.out.println("Area is " + figref.area());

}
```

}

Output:

```
D:\>javac AbstractAreas.java

D:\>java AbstractAreas
Inside Area for Rectangle.
Area is 45.0
Inside Area for Triangle.
Area is 40.0

D:\>
```

### Differences Between Compile-time and Run-time Polymorphism

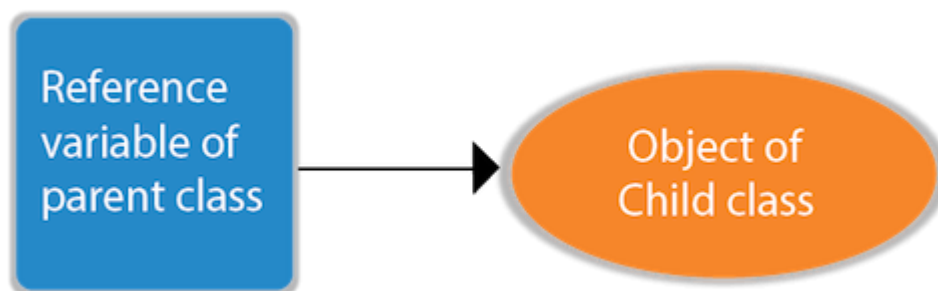
- There are two types of polymorphism in java:
- 1) **Static Polymorphism** also known as compile time polymorphism.  
2) **Dynamic Polymorphism** also known as runtime polymorphism.
- **Compile Time Polymorphism:**
- Whenever an object is bound with their functionality at the compile-time, this is known as the compile-time polymorphism.
- At compile-time, java knows which method to call by checking the method signatures.

- **this is called compile-time polymorphism or static or early binding.**
- **Compile-time polymorphism** is achieved through method overloading.
- Method Overloading says you can have more than one function with the same name in one class having a different prototype. Function overloading is one of the ways to achieve polymorphism but it depends on technology that which type of polymorphism we adopt. In java, we achieve function overloading at compile-Time.
- **Run-Time Polymorphism:**
- **Whenever an object is bound with the functionality at run time, this is known as runtime polymorphism.**
- **The runtime polymorphism can be achieved by method overriding.**
- **Java virtual machine determines the proper method to call at the runtime, not at the compile time.**
- It is also called dynamic or late binding.
- Method overriding says child class has the same method as declared in the parent class. It means if child class provides the specific implementation of the method that has been provided by one of its parent class then it is known as method overriding.

Sr.No	Compile Time Polymorphism	Runtime Polymorphism
1.	We can explain compile-time polymorphism through method overloading. Compile-time polymorphism allows us to have more than one method share the same name with different signatures and different return types.	We can explain run-time polymorphism through method overriding. Run-time polymorphism is allied in different classes but allows us to have the same method with the same signature name.
2	In this, the call is determined by the compiler.	In this, the call is not determined by the compiler.
3	The method is executed quite earlier at the compile-time, and that's why it provides fast execution.	The method is executed at the run-time, and that's why it provides slow execution.
4	This polymorphism is also known as early binding, overloading, and static binding.	This polymorphism is also known as late binding, dynamic binding, and overriding.
5	It is obtained by operator overloading and function overloading.	It is obtained by pointers and virtual functions.
6	It is less manageable as it performs at compile time.	It is more flexible as it executes at run time.

## Runtime Polymorphism (or ) 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.
- Let's first understand the upcasting before Runtime Polymorphism.
- **Note : Upcasting :**
- If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:



### **Example : upcasting**

```
class A
```

```
{
```

```
}
```

```
class B extends A
```

```
{
```

```
}
```

```
A a:
```

```
B b=new B();
```

```
a=b; //upcasting
```

For upcasting, we can use the reference variable of class type.

### **Example:**

```
class A
```

```
{
```

```
void callme()
```

```
{
```

```
System.out.println("Inside A's callme method");
```

```
}
```

```
}
```

```
class B extends A
{
    // override callme()
    void callme()
    {
        System.out.println("Inside B's callme method");
    }
}

class C extends A
{
    // override callme()
    void callme()
    {
        System.out.println("Inside C's callme method");
    }
}

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



```
{  
A a = new A(); // object of type A  
B b = new B(); // object of type B  
C c = new C(); // object of type C  
A r; // obtain a reference of type A  
r = a; // r refers to an A object  
r.callme(); // calls A's version of callme()  
r = b; // r refers to a B object  
r.callme(); // calls B's version of callme()  
r = c; // r refers to a C object  
r.callme(); // calls C's version of callme()  
}  
}
```

**The output from the program is shown here:**

Inside A's callme method

Inside B's callme method

Inside C's callme method

## Usage of final keyword

- The final keyword in java is used to restrict the user. The java final keyword can be used in many context. Final can be:
  1. Variable
  2. Method
  3. class
- **final variable:** If we make any variable as final, we cannot change the value of final variable(It will be constant).

- **Example :**

```
class base
{
    final int a=100;//final variable
    void run()
    {
        a=200;
    }
    public static void main(String args[])
    {
        base obj=new base();
        obj.run();
    }
}
```

## The output

```
D:\>javac base.java
base.java:6: cannot assign a value to final variable a
    a=200;
    ^
1 error
```

- **2) final method** :If we make any method as final, we cannot override it.

- **Example:**

```
class base
```

```
{
```

```
    final void run()
```

```
{
```

```
    System.out.println("running");
```

```
}
```

```
}
```

```
class derived extends base
```

```
{
```

```
    void run()
```

```
{
```

```
    System.out.println("running safely with 100kmph");
```

```
}
```

```
public static void main(String args[])
{
    derived d= new derived();
    d.run();
}
}
```

### Output:

```
D:\>javac base.java
base.java:11: run() in derived cannot override run() in base; overridden method is final
    void run()
        ^
1 error
```

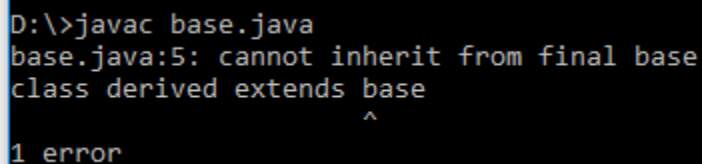
- **final class** :If we make any class as final, we cannot extend it.
- **Example:**

```
final class base
{
    void run1()
    {
        System.out.println("running safely with 50kmph");
    }
}

class derived extends base
```

```
{
void run2()
{
System.out.println("running safely with 100kmph");
}
public static void main(String args[])
{
derived d= new derived();
d.run();
}
}
```

**Output:**

A screenshot of a terminal window showing a Java compilation error. The text is as follows:

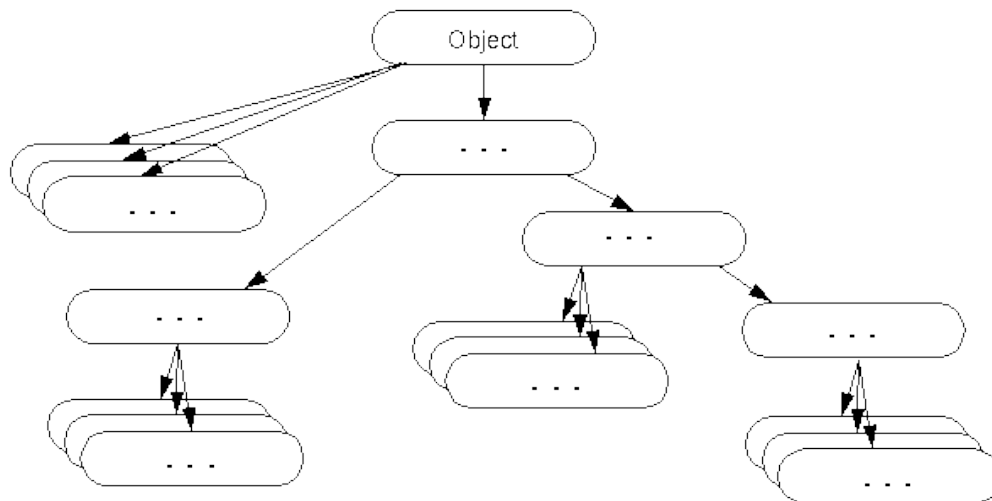
```
D:\>javac base.java
base.java:5: cannot inherit from final base
class derived extends base
                ^
1 error
```

## **Object class**

- There is one special class, **Object**, defined by Java.
- **Object** class is present in **java.lang** package. Every class in Java is directly or indirectly derived from the **Object** class. If a class does not extend any other class then it is a direct child class

of **Object** and if extends another class then it is indirectly derived. Therefore the Object class methods are available to all Java classes. Hence Object class acts as a root of the inheritance hierarchy in any Java Program.

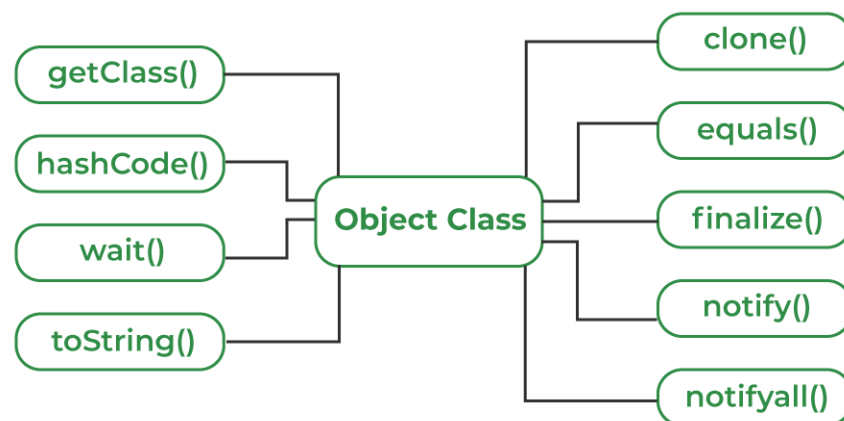
- The Object class is the parent class of all the classes in java by default.
- In other words, it is the topmost class of java.
- All other classes are subclasses of Object. That is, Object is a superclass of all other classes. This means that a reference variable of type Object can refer to an object of any other class.



- **Object** class is present in **java.lang** package. Every class in Java is directly or indirectly derived from the **Object** class. If a class does not extend any other class then it is a direct child class of **Object** and if extends another class then it is indirectly derived.

Therefore the Object class methods are available to all Java classes. Hence Object class acts as a root of the inheritance hierarchy in any Java Program.

### Methods of Object class:



The Object class provides many methods. They are as follows:

- protected Object clone() - Used to create and return a copy of this object.
- boolean equals(Object obj) - Used to indicate whether some other object is "equal to" this one.
- protected void finalize() - garbage collector calls this method on an object when it determines that there are no more references to the object.
- Class<?>getClass() - Used to get the runtime class of this Object.
- int hashCode() - Used to get a hash code value for the object.

- `void notify()` - Used to wake up a single thread that is waiting on this object's monitor.
- `void notifyAll()` - Used to wake up all threads that are waiting on this object's monitor.
- `String toString()` - Used to get a string representation of the object.
- `void wait()` - marks the current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object.
- `void wait(long timeout)` - marks the current thread to wait until either another thread invokes the `notify()` method or the `notifyAll()` method for this object, or a specified amount of time has elapsed.
- `void wait(long timeout, int nanos)` - marks the current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

### **Example Programs:**