

UNIT II INHERITANCE AND INTERFACES

Inheritance – Super classes- sub classes –Protected members – constructors in sub classes- the Object class – abstract classes and methods- final methods and classes – Interfaces – defining an interface, implementing interface, differences between classes and interfaces and extending interfaces - Object cloning -inner classes, Array Lists – Strings

INHERITANCE

Inheritance can be defined as the procedure or mechanism of acquiring all the properties and behaviors of one class to another, i.e., acquiring the properties and behavior of child class from the parent class.

When one object acquires all the properties and behaviours of another object, it is known as inheritance. Inheritance represents the **IS-A relationship**, also known as *parent-child* relationship.

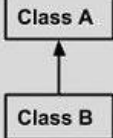
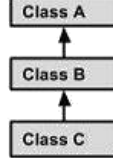
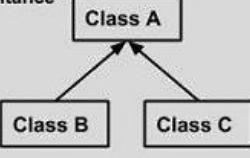
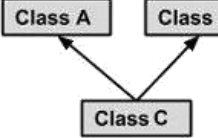
Uses of inheritance in java

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

Types of inheritance in java: single, multilevel and hierarchical inheritance. Multiple and hybrid inheritance is supported through interface only.

Syntax:

```
class subClass extends superClass
{
    //methods and fields
}
```

<p>Single Inheritance</p>  <pre> graph BT B[Class B] --> A[Class A] </pre>	<pre> public class A { } public class B extends A { } </pre>
<p>Multi Level Inheritance</p>  <pre> graph BT C[Class C] --> B[Class B] B --> A[Class A] </pre>	<pre> public class A {} public class B extends A {.....} public class C extends B {.....} </pre>
<p>Hierarchical Inheritance</p>  <pre> graph BT B[Class B] --> A[Class A] C[Class C] --> A </pre>	<pre> public class A {} public class B extends A {.....} public class C extends A {.....} </pre>
<p>Multiple Inheritance</p>  <pre> graph BT C[Class C] --> A[Class A] C --> B[Class B] </pre>	<pre> public class A {} public class B {.....} public class C extends A,B { } // Java does not support mutiple Inheritance </pre>

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 previous class.

SINGLE INHERITANCE

In Single Inheritance one class extends another class (one class only).

Example:

```
public class ClassA
{
    public void dispA()
    {
        System.out.println("disp() method of ClassA");
    }
}
public class ClassB extends ClassA
{
    public void dispB()
    {
        System.out.println("disp() method of ClassB");
    }
    public static void main(String args[])
    {
        //Assigning ClassB object to ClassB reference
        ClassB b = new ClassB();
        //call dispA() method of ClassA
        b.dispA();
        //call dispB() method of ClassB
        b.dispB();
    }
}
```

Output :

```
disp() method of ClassA
disp() method of ClassB
```

MULTILEVEL INHERITANCE

In Multilevel Inheritance, one class can inherit from a derived class. Hence, the derived class becomes the base class for the new class.

Example:

```
public class ClassA
{
    public void dispA()
    {
        System.out.println("disp() method of ClassA");
    }
}
public class ClassB extends ClassA
{
    public void dispB()
    {
        System.out.println("disp() method of ClassB");
    }
}
public class ClassC extends ClassB
{
    public void dispC()
    {
        System.out.println("disp() method of ClassC");
    }
    public static void main(String args[])
    {
        // Assigning ClassC object to ClassC reference
        ClassC c = new ClassC();
        // call dispA() method of ClassA
        c.dispA();
        // call dispB() method of ClassB
        c.dispB();
        // call dispC() method of ClassC
        c.dispC();
    }
}
```

Output :

```
disp() method of ClassA
disp() method of ClassB
disp() method of ClassC
```

HIERARCHICAL INHERITANCE

In Hierarchical Inheritance, one class is inherited by many sub classes.

Example:

```
public class ClassA
{
    public void dispA()
    {
        System.out.println("disp() method of ClassA");
    }
}
public class ClassB extends ClassA
{
    public void dispB()
    {
        System.out.println("disp() method of ClassB");
    }
}
public class ClassC extends ClassA
{
    public void dispC()
    {
        System.out.println("disp() method of ClassC");
    }
}
public class ClassD extends ClassA
{
    public void dispD()
    {
        System.out.println("disp() method of ClassD");
    }
}
public class HierarchicalInheritanceTest
{
    public static void main(String args[])
    {
        // Assigning ClassB object to ClassB reference
        ClassB b = new ClassB();
        // call dispB() method of ClassB
        b.dispB();
        // call dispA() method of ClassA
        b.dispA();
        // Assigning ClassC object to ClassC reference
        ClassC c = new ClassC();
        // call dispC() method of ClassC
        c.dispC();
    }
}
```

```

//call dispA() method of ClassA
c.dispA();

//Assigning ClassD object to ClassD reference
ClassD d = new ClassD();
//call dispD() method of ClassD
d.dispD();
//call dispA() method of ClassA
d.dispA();
}
}

```

Output :

```

disp() method of ClassB
disp() method of ClassA
disp() method of ClassC
disp() method of ClassA
disp() method of ClassD
disp() method of ClassA

```

Hybrid Inheritance is the combination of both Single and Multiple Inheritance. Again Hybrid inheritance is also not directly supported in Java only through interface we can achieve this. Flow diagram of the Hybrid inheritance will look like below. As you can ClassA will be acting as the Parent class for ClassB & ClassC and ClassB & ClassC will be acting as Parent for ClassD.

Multiple Inheritance is nothing but one class extending more than one class. Multiple Inheritance is basically not supported by many Object Oriented Programming languages such as Java, Small Talk, C# etc.. (C++ Supports Multiple Inheritance). As the Child class has to manage the dependency of more than one Parent class. But you can achieve multiple inheritance in Java using Interfaces.

“super” KEYWORD***Usage of super keyword***

1. super() invokes the constructor of the parent class.
2. super.variable_name refers to the variable in the parent class.
3. super.method_name refers to the method of the parent class.

1. super() invokes the constructor of the parent class

super() will invoke the constructor of the parent class. Even when you don't add **super()** keyword the compiler will add one and will invoke the [Parent Class constructor](#).

Example:

```

class ParentClass
{
    ParentClass()

```

```
    {
        System.out.println("Parent Class default Constructor");
    }
}
public class SubClass extends ParentClass
{
    SubClass()
    {
        System.out.println("Child Class default Constructor");
    }
    public static void main(String args[])
    {
        SubClass s = new SubClass();
    }
}
```

Output:

Parent Class default Constructor
Child Class default Constructor

Even when we add explicitly also it behaves the same way as it did before.

```
class ParentClass
{
    public ParentClass()
    {
        System.out.println("Parent Class default Constructor");
    }
}
public class SubClass extends ParentClass
{
    SubClass()
    {
        super();
        System.out.println("Child Class default Constructor");
    }
    public static void main(String args[])
    {
        SubClass s = new SubClass();
    }
}
```

Output:

Parent Class default Constructor
Child Class default Constructor

You can also call the parameterized constructor of the Parent Class. For example, **super(10)** will call parameterized constructor of the Parent class.

```
class ParentClass
{
    ParentClass()
    {
        System.out.println("Parent Class default Constructor called");
    }
    ParentClass(int val)
    {
        System.out.println("Parent Class parameterized Constructor, value: "+val);
    }
}
public class SubClass extends ParentClass
{
    SubClass()
    {
        super();//Has to be the first statement in the constructor
        System.out.println("Child Class default Constructor called");
    }
    SubClass(int val)
    {
        super(10);
        System.out.println("Child Class parameterized Constructor, value: "+val);
    }
    public static void main(String args[])
    {
        //Calling default constructor
        SubClass s = new SubClass();
        //Calling parameterized constructor
        SubClass s1 = new SubClass(10);
    }
}
```

Output

```
Parent Class default Constructor called
Child Class default Constructor called
Parent Class parameterized Constructor, value: 10
Child Class parameterized Constructor, value: 10
```

2. super.variable_name refers to the variable in the parent class

When we have the same variable in both parent and subclass

```
class ParentClass
{
```

```
        int val=999;
    }
    public class SubClass extends ParentClass
    {
        int val=123;

        void disp()
        {
            System.out.println("Value is : "+val);
        }

        public static void main(String args[])
        {
            SubClass s = new SubClass();
            s.disp();
        }
    }
```

Output

Value is : 123

This will call only the **val** of the sub class only. Without **super** keyword, you cannot call the **val** which is present in the Parent Class.

```
class ParentClass
{
    int val=999;
}
public class SubClass extends ParentClass
{
    int val=123;

    void disp()
    {
        System.out.println("Value is : "+super.val);
    }

    public static void main(String args[])
    {
        SubClass s = new SubClass();
        s.disp();
    }
}
```

Output

Value is : 999

3. super.method_name refers to the method of the parent class

When you override the Parent Class method in the Child Class without super keywords support you will not be able to call the Parent Class method. Let's look into the below example

```
class ParentClass
{
    void disp()
    {
        System.out.println("Parent Class method");
    }
}
public class SubClass extends ParentClass
{
    void disp()
    {
        System.out.println("Child Class method");
    }
    void show()
    {
        disp();
    }
    public static void main(String args[])
    {
        SubClass s = new SubClass();
        s.show();
    }
}
```

Output:

Child Class method

Here we have overridden the **Parent Class disp()** method in the SubClass and hence **SubClass disp()** method is called. If we want to call the **Parent Class disp()** method also means then we have to use the super keyword for it.

```
class ParentClass
{
    void disp()
    {
        System.out.println("Parent Class method");
    }
}
public class SubClass extends ParentClass
{

```

```

void disp()
{
    System.out.println("Child Class method");
}

void show()
{
    //Calling SubClass disp() method
    disp();
    //Calling ParentClass disp()
    method super.disp();
}
public static void main(String args[])
{
    SubClass s = new SubClass();
    s.show();
}
}

```

Output

Child Class method
Parent Class method

When there is no method overriding then by default **Parent Class disp()** method will be called.

```

class ParentClass
{
    public void disp()
    {
        System.out.println("Parent Class method");
    }
}
public class SubClass extends ParentClass
{
    public void show()
    {
        disp();
    }
    public static void main(String args[])
    {
        SubClass s = new SubClass();
        s.show(); }
}

```

Output:

Parent Class method

The Object Class

There is one special class, **Object**, defined by 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. Also, since arrays are implemented as classes, a variable of type **Object** can also refer to any array. **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 <i>object</i>)	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 <i>milliseconds</i>) void wait(long <i>milliseconds</i> , int <i>nanoseconds</i>)	Waits on another thread of execution.

The methods **getClass()**, **notify()**, **notifyAll()**, and **wait()** are declared as **final**. You may override the others. These methods are described elsewhere in this book. However, notice two methods now: **equals()** and **toString()**. The **equals()** method compares two objects. It returns **true** if the objects are equal, and **false** otherwise. The precise definition of equality can vary, depending on the type of objects being compared. The **toString()** method returns a string that contains a description of the object on which it is called. Also, this method is automatically called when an object is output using **println()**. Many classes override this method. Doing so allows them to tailor a description specifically for the types of objects that they create.

ABSTRACT CLASS

A class that is declared with abstract keyword, is known as abstract class in java. It can have abstract and non-abstract methods (method with body). Abstraction is a process of hiding the implementation details and showing only functionality to the user. Abstraction lets you focus on what the object does instead of how it does it. It needs to be extended and its method implemented. It cannot be instantiated.

Example abstract class:

```
abstract class A{
```

abstract method:

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

```
abstract void printStatus(); //no body and abstract
```

In this example, Shape is the abstract class, its implementation is provided by the Rectangle and Circle classes.

If you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

Example1:

File: TestAbstraction1.java

```
abstract class Shape{
abstract void draw();
}
```

//In real scenario, implementation is provided by others i.e. unknown by end

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

```
class Circle1 extends Shape{
void draw(){System.out.println("drawing circle");}
}
```

//In real scenario, method is called by programmer or user

```
class TestAbstraction1{
public static void main(String args[]){
Shape s=new Circle1();//In real scenario, object is provided through method e.g. getShape() method
s.draw();
}
}
```

Output:

drawing circle

Abstract class having constructor, data member, methods

An abstract class can have data member, abstract method, method body, constructor and even main() method.

Example2:

File: TestAbstraction2.java

//example of abstract class that have method body

```
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 TestAbstraction2{
public static void main(String args[]){
    Bike obj = new Honda();
    obj.run();
    obj.changeGear();
}
}
```

Output:

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

The abstract class can also be used to provide some implementation of the interface. In such case, the end user may not be forced to override all the methods of the interface.

Example3:

```
interface A{
void a();
void b();
void c();
void d();
}
abstract class B implements A{
public void c(){System.out.println("I am c");}
}
class M extends B{
public void a(){System.out.println("I am a");}
public void b(){System.out.println("I am b");}
public void d(){System.out.println("I am d");}
}
class Test5{
public static void main(String args[]){
A a=new M();
a.a();
a.b();
a.c();
a.d();
}}
```

Output:

```
I am a
I am b
I am c
I am d
```

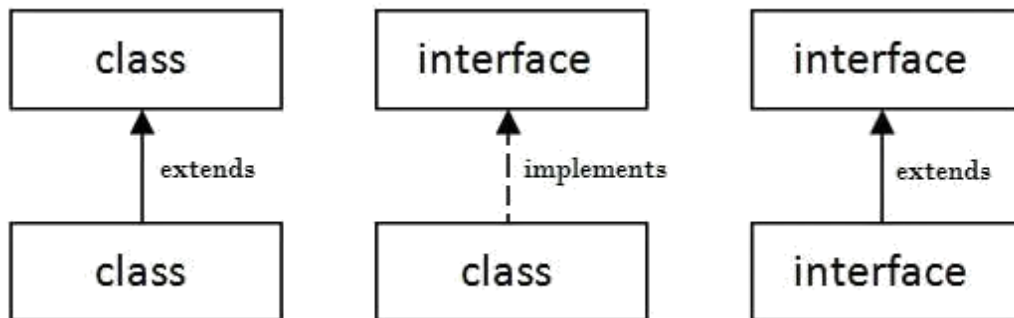
INTERFACE IN JAVA

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 and multiple inheritance.

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

Syntax:

```
interface <interface_name>
{
    // declare constant fields
    // declare methods that abstract
    // by default.
}
```

Relationship between classes and interfaces**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

Example: interface

Drawable

```
{
void draw();
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
```

```
class Circle implements Drawable{
public void draw(){System.out.println("drawing circle");}
}
```

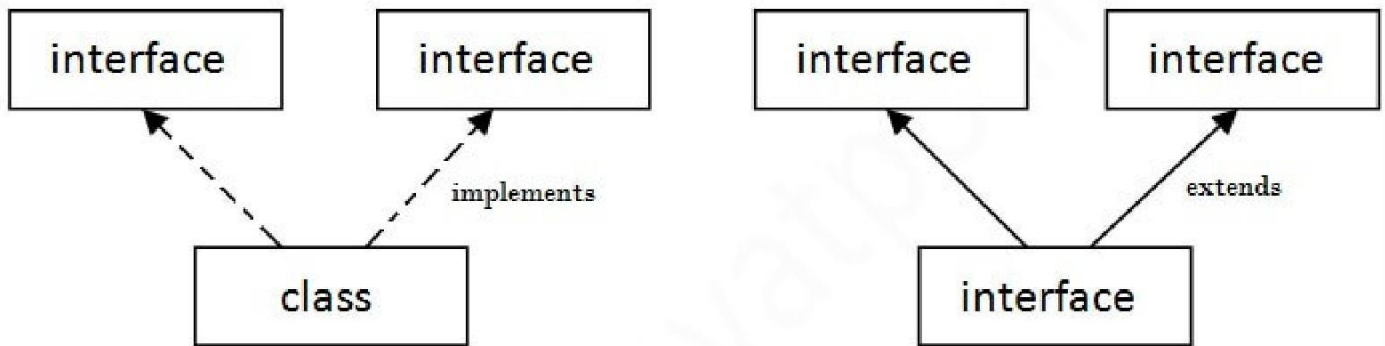
```
class TestInterface1{
public static void main(String args[]){
Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDrawable()
d.draw();
}
}
```

Output:

drawing circle

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.



Example: interface

```
Printable{ void
print();
}
```

```
interface Showable{
void show();
}
```

```
class A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}
```

```
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
}
}
```

Output:

Hello
Welcome
Interface inheritance

A class implements 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

Nested Interface in Java

An interface can have another interface i.e. known as nested interface.

```
interface printable{
    void print();
    interface MessagePrintable{
        void msg();
    }
}
```


Key points to remember about interfaces:

- 1) We can't instantiate an interface in java. That means we cannot create the object of an interface
- 2) Interface provides full abstraction as none of its methods have body. On the other hand abstract class provides partial abstraction as it can have abstract and concrete(methods with body) methods both.
- 3) "implements" keyword is used by classes to implement an interface.
- 4) While providing implementation in class of any method of an interface, it needs to be mentioned as public.
- 5) Class that implements any interface must implement all the methods of that interface, else the class should be declared abstract.
- 6) Interface cannot be declared as private, protected or transient.
- 7) All the interface methods are by default abstract and public.
- 8) Variables declared in interface are public, static and final by default.

interface Try

```
{  
    int a=10; public  
    int a=10;  
    public static final int a=10;  
    final int a=10;  
    static int a=0;  
}
```

All of the above statements are identical.

- 9) Interface variables must be initialized at the time of declaration otherwise compiler will throw an error.

interface Try

```
{  
    int x;//Compile-time error  
}
```

Above code will throw a compile time error as the value of the variable x is not initialized at the time of declaration.

- 10) Inside any implementation class, you cannot change the variables declared in interface because by default, they are public, static and final. Here we are implementing the interface "Try" which has a variable x. When we tried to set the value for variable x we got compilation error as the variable x is public static **final** by default and final variables can not be re-initialized.

class Sample implements Try

```
{  
    public static void main(String args[])  
    {  
        x=20; //compile time error  
    }  
}
```

- 11) An interface can extend any interface but cannot implement it. Class implements interface and interface extends interface.

12) A class can implement any number of interfaces.

13) If there are two or more same methods in two interfaces and a class implements both interfaces, implementation of the method once is enough.

```
interface A
{
    public void aaa();
}
interface B
{
    public void aaa();
}
class Central implements A,B
{
    public void aaa()
    {
        //Any Code here
    }
    public static void main(String args[])
    {
        //Statements
    }
}
```

14) A class cannot implement two interfaces that have methods with same name but different return type.

```
interface A
{
    public void aaa();
}
interface B
{
    public int aaa();
}
class Central implements A,B
{
    public void aaa() // error
    {
    }
    public int aaa() // error
    {
    }
    public static void main(String args[])
    {
    }
}
```

```

}
}
15) Variable names conflicts can be resolved by interface
name. interface A
{
    int x=10;
}
interface B
{
    int x=100;
}
class Hello implements A,B
{
    public static void Main(String args[])
    {
        System.out.println(x);
        System.out.println(A.x);
        System.out.println(B.x);
    }
}
}

```

Advantages of interface in java:

- Without bothering about the implementation part, we can achieve the security of implementation
- In java, **multiple inheritance** is not allowed, however you can use interface to make use of it as you can implement more than one interface.

DIFFERENCE BETWEEN ABSTRACT CLASS AND INTERFACE

ABSTRACT CLASS	INTERFACE
1) Abstract class can have abstract and non-abstract methods.	Interface can have only abstract methods. Since Java 8, it can have default and static methods also.
2) Abstract class doesn't support multiple inheritance.	Interface supports multiple inheritance.
3) Abstract class can have final, non-final, static and non-static variables.	Interface has only static and final variables.
4) Abstract class can provide the implementation of interface.	Interface can't provide the implementation of abstract class.
5) The abstract keyword is used to declare abstract class.	The interface keyword is used to declare interface.

6) An abstract class can extend another Java class and implement multiple Java interfaces.	An interface can extend another Java interface only.
7) An abstract class can be extended using keyword extends.	An interface class can be implemented using keyword implements
8) A Java abstract class can have class members like private, protected, etc.	Members of a Java interface are public by default.
9) Example: <pre>public abstract class Shape{ public abstract void draw(); }</pre>	Example: <pre>public interface Drawable{ void draw(); }</pre>

FINAL KEYWORD

Final keyword can be used along with **variables, methods and classes.**

- 1) **final variable**
- 2) **final method**
- 3) **final class**

1. Java final variable

A **final variable** is a variable whose value **cannot** be changed at anytime once assigned, it remains as a constant forever.

Example:

```
public class Travel
{
    final int SPEED=60;
    void increaseSpeed(){
        SPEED=70;
    }
    public static void main(String args[])
    {
        Travel t=new Travel();
        t.increaseSpeed();
    }
}
```

Output :

Exception in thread "main" java.lang.Error: Unresolved compilation problem:
 The final field Travel.SPEED cannot be assigned

The above code will give you Compile time error, as we are trying to change the value of a final variable '**SPEED**'.

2. Java final method

When you declare a method as **final**, then it is called as **final method**. A **final method cannot be overridden**.

```
package com.javainterviewpoint;

class Parent
{
    public final void disp()
    {
        System.out.println("disp() method of parent class");
    }
}
public class Child extends Parent
{
    public void disp()
    {
        System.out.println("disp() method of child class");
    }
    public static void main(String args[])
    {
        Child c = new Child();
        c.disp();
    }
}
```

Output : We will get the below error as we are overriding the **disp()** method of the **Parent** class. Exception in thread "main" java.lang.VerifyError: class com.javainterviewpoint.Child overrides final method disp.()

```
    at java.lang.ClassLoader.defineClass1(Native Method)
    at java.lang.ClassLoader.defineClass(Unknown Source)
    at java.security.SecureClassLoader.defineClass(Unknown Source)
    at java.net.URLClassLoader.defineClass(Unknown Source)
    at java.net.URLClassLoader.access$100(Unknown Source)
    at java.net.URLClassLoader$1.run(Unknown Source)
    at java.net.URLClassLoader$1.run(Unknown Source)
```

3. Java final class

A final class cannot be extended (cannot be subclassed), let's take a look into the below example package com.javainterviewpoint;

```
final class Parent
{
}
public class Child extends Parent
{
```

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

Output :

We will get the compile time error like *"The type Child cannot subclass the final class Parent"*
Exception in thread "main" java.lang.Error: Unresolved compilation problem

OBJECT CLONING

The **object cloning** is a way to create exact copy of an object. The clone() method of Object class is used to clone an object.

The **java.lang.Cloneable interface** must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates **CloneNotSupportedException**.

The **clone() method** is defined in the Object class.

Syntax of the clone() method:

protected Object clone() throws CloneNotSupportedException

The **clone() method** saves the extra processing task for creating the exact copy of an object. If we perform it by using the new keyword, it will take a lot of processing time to be performed that is why we use object cloning.

Advantage of Object cloning

- You don't need to write lengthy and repetitive codes. Just use an abstract class with a 4- or 5-line long clone() method.
- It is the easiest and most efficient way for copying objects, especially if we are applying it to an already developed or an old project. Just define a parent class, implement Cloneable in it, provide the definition of the clone() method and the task will be done.
- Clone() is the fastest way to copy array.

Disadvantage of Object cloning

- To use the Object.clone() method, we have to change a lot of syntaxes to our code, like implementing a Cloneable interface, defining the clone() method and handling CloneNotSupportedException, and finally, calling Object.clone() etc.
- We have to implement cloneable interface while it doesn't have any methods in it. We just have to use it to tell the JVM that we can perform clone() on our object.
- Object.clone() is protected, so we have to provide our own clone() and indirectly call Object.clone() from it.
- Object.clone() doesn't invoke any constructor so we don't have any control over object construction.
- If you want to write a clone method in a child class then all of its superclasses should define the clone() method in them or inherit it from another parent class. Otherwise, the super.clone() chain will fail.

- Object.clone() supports only shallow copying but we will need to override it if we need deep cloning.

Example of clone() method (Object cloning)

```
class Student implements Cloneable{
int rollno;
String name;
Student(int rollno,String
name){ this.rollno=rollno;
this.name=name;
}
public Object clone()throws CloneNotSupportedException{
return super.clone();
}
public static void main(String args[]){
try{
Student s1=new Student(101,"amit");
Student s2=(Student)s1.clone();
System.out.println(s1.rollno+" "+s1.name);
System.out.println(s2.rollno+" "+s2.name);
}
catch(CloneNotSupportedException c){}
}
}
```

Output:

```
101 amit
101 amit
```

INNER CLASSES

Inner class means one class which is a member of another class. There are basically four types of inner classes in java.

- 1) Nested Inner class
- 2) Method Local inner classes
- 3) Anonymous inner classes
- 4) Static nested classes

Nested Inner class

Nested Inner class can access any private instance variable of outer class. Like any other instance variable, we can have access modifier private, protected, public and default modifier. Like class, interface can also be nested and can have access specifiers.

Example:

```
class Outer {
// Simple nested inner class
class Inner {
```

```
    public void show() {
        System.out.println("In a nested class method");
    }
}
}
class Main {
    public static void main(String[] args) {
        Outer.Inner in = new Outer().new Inner();
        in.show();
    }
}
```

Output:

In a nested class method

Method Local inner classes

Inner class can be declared within a method of an outer class. In the following example, Inner is an inner class in outerMethod().

Example:

```
class Outer {
    void outerMethod() {
        System.out.println("inside outerMethod");
        // Inner class is local to outerMethod()
        class Inner {
            void innerMethod() {
                System.out.println("inside innerMethod");
            }
        }
        Inner y = new Inner();
        y.innerMethod();
    }
}
class MethodDemo {
    public static void main(String[] args) {
        Outer x = new Outer();
        x.outerMethod();
    }
}
```

Output:

Inside outerMethod
Inside innerMethod

Static nested classes

Static nested classes are not technically an inner class. They are like a static member of outer

class.

Example:

```
class Outer {
    private static void outerMethod() {
        System.out.println("inside outerMethod");
    }
    // A static inner class
    static class Inner {
        public static void main(String[] args) {
            System.out.println("inside inner class Method");
            outerMethod();
        }
    }
}
```

Output:

inside inner class Method
inside outerMethod

Anonymous inner classes

Anonymous inner classes are declared without any name at all. They are created in two ways.

a) As subclass of specified type

```
class Demo {
    void show() {
        System.out.println("i am in show method of super class");
    }
}
class Flavor1Demo {
    // An anonymous class with Demo as base class
    static Demo d = new Demo() {
        void show() {
            super.show();
            System.out.println("i am in Flavor1Demo class");
        }
    };
    public static void main(String[] args){
        d.show();
    }
}
```

Output:

i am in show method of super class
i am in Flavor1Demo class

In the above code, we have two class Demo and Flavor1Demo. Here demo act as super class and anonymous class acts as a subclass, both classes have a method show(). In anonymous

class show() method is overridden.

b) As implementer of the specified interface

Example:

```
class Flavor2Demo {
    // An anonymous class that implements Hello interface
    static Hello h = new Hello() {
        public void show() {
            System.out.println("i am in anonymous class");
        }
    };
    public static void main(String[] args) {
        h.show();
    }
}
interface Hello {
    void show(); }
```

Output:

i am in anonymous class

In above code we create an object of anonymous inner class but this anonymous inner class is an implementer of the interface Hello. Any anonymous inner class can implement only one interface at one time. It can either extend a class or implement interface at a time.

STRINGS IN JAVA

In java, string is basically an object that represents sequence of char values. **Java String** provides a lot of concepts that can be performed on a string such as compare, concat, equals, split, length, replace, compareTo, intern, substring etc.

In java, **string objects are immutable**. Immutable simply means unmodifiable or unchangeable.

```
String s="javatpoint";
```

There are two ways to create String object:

1. By string literal
2. By new keyword

1) String Literal

Java String literal is created by using double quotes. For Example:

```
String s="welcome";
```

2) By new keyword

```
String s=new String("Welcome");
```

String methods:		
1.	char charAt(int index)	returns char value for the particular index
2.	int length()	returns string length
3.	static String format(String format, Object... args)	returns formatted string
4.	static String format(Locale l, String format, Object... args)	returns formatted string with given locale
5.	String substring(int beginIndex)	returns substring for given begin index
6.	String substring(int beginIndex, int endIndex)	returns substring for given begin index and end index
7.	boolean contains(CharSequence s)	returns true or false after matching the sequence of char value
8.	static String join(CharSequence delimiter, CharSequence... elements)	returns a joined string
9.	static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)	returns a joined string
10.	boolean equals(Object another)	checks the equality of string with object
11.	boolean isEmpty()	checks if string is empty
12.	String concat(String str)	concatinates specified string
13.	String replace(char old, char new)	replaces all occurrences of specified char value
14.	String replace(CharSequence old, CharSequence new)	replaces all occurrences of specified CharSequence
15.	static String equalsIgnoreCase(String another)	compares another string. It doesn't check case.
16.	String[] split(String regex)	returns splitted string matching regex
17.	String[] split(String regex, int limit)	returns splitted string matching regex and limit
18.	String intern()	returns interned string
19.	int indexOf(int ch)	returns specified char value index
20.	int indexOf(int ch, int fromIndex)	returns specified char value index starting with given index
21.	int indexOf(String substring)	returns specified substring index
22.	int indexOf(String substring, int fromIndex)	returns specified substring index starting with given index
23.	String toLowerCase()	returns string in lowercase.
24.	String toLowerCase(Locale l)	returns string in lowercase using specified locale.
25.	String toUpperCase()	returns string in uppercase.
26.	String toUpperCase(Locale l)	returns string in uppercase using specified locale.
27.	String trim()	removes beginning and ending spaces of this string.
28.	static String valueOf(int value)	converts given type into string. It is overloaded.

Example:

```

public class stringmethod
{
    public static void main(String[] args)
    {
        String string1 = new String("hello");
        String string2 = new String("hello");
        if (string1 == string2)
        {
            System.out.println("string1= "+string1+" string2= "+string2+" are equal");
        }
        else
        {
            System.out.println("string1= "+string1+" string2= "+string2+" are Unequal");
        }
        System.out.println("string1 and string2 is=
        "+string1.equals(string2)); String a="information";
        System.out.println("Uppercase of String a is= "+a.toUpperCase());
        String b="technology";
        System.out.println("Concatenation of object a and b is= "+a.concat(b));
        System.out.println("After concatenation Object a is= "+a.toString());
        System.out.println("\"Joseph's\" is the greatest\ \ college in chennai");
        System.out.println("Length of Object a is= "+a.length());
        System.out.println("The third character of Object a is= "+a.charAt(2));
        StringBuffer n=new StringBuffer("Technology");
        StringBuffer m=new StringBuffer("Information");
        System.out.println("Reverse of Object n is= "+n.reverse());
        n= new StringBuffer("Technology");
        System.out.println("Concatenation of Object m and n is= "+m.append(n));
        System.out.println("After concatenation of Object m is= "+m);
    }
}

```

Output:

```

string1= hello string2= hello are Unequal
string1 and string2 is= true
Uppercase of String a is= INFORMATION
Concatenation of object a and b is= informationtechnology
After concatenation Object a is= information
"Joseph's" is the greatest\ college in chennai
Length of Object a is= 11
The third character of Object a is= f
Reverse of Object n is= ygolnhceT
Concatenation of Object m and n is= InformationTechnology

```

After concatenation of Object m is= InformationTechnology

Java ArrayList class

Java ArrayList class uses a dynamic array for storing the elements. It inherits AbstractList class and implements List interface.

The important points about Java ArrayList class are:

- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non synchronized.
- Java ArrayList allows random access because array works at the index basis.
- In Java ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

ArrayList class declaration Syntax:

public class ArrayList<E> **extends** AbstractList<E> **implements** List<E>, RandomAccess, Cloneable, Serializable

Constructors of Java ArrayList

CONSTRUCTOR	DESCRIPTION
ArrayList()	It is used to build an empty array list.
ArrayList(Collection c)	It is used to build an array list that is initialized with the elements of the collection c.
ArrayList(int capacity)	It is used to build an array list that has the specified initial capacity.

Methods of Java ArrayList

METHOD	DESCRIPTION
void add(int index, Object element)	It is used to insert the specified element at the specified position index in a list.
boolean addAll(Collection c)	It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
void clear()	It is used to remove all of the elements from this list.
int lastIndexOf(Object o)	It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
Object[] toArray()	It is used to return an array containing all of the elements in this list in the correct order.
Object[] toArray(Object[] a)	It is used to return an array containing all of the elements in this list in the correct order.
boolean add(Object o)	It is used to append the specified element to the end of a list.
boolean addAll(int index, Collection c)	It is used to insert all of the elements in the specified collection into this list, starting at the specified position.
Object clone()	It is used to return a shallow copy of an ArrayList.
int indexOf(Object o)	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.

Java ArrayList Example: Book

Example:

```
import java.util.*;
class Book {
int id;
String name,author,publisher;
int quantity;
public Book(int id, String name, String author, String publisher, int quantity) {
    this.id = id;
    this.name = name;
    this.author = author;
    this.publisher = publisher;
    this.quantity = quantity;
}
}

public class ArrayListExample {
public static void main(String[] args) {
    //Creating list of Books
    List<Book> list=new ArrayList<Book>();
    //Creating Books
    Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
    Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
    Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
    //Adding Books to list
    list.add(b1);
    list.add(b2);
    list.add(b3);
    //Traversing list
    for(Book b:list){
        System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
    }
}
}
```

Output:

```
101 Let us C Yashwant Kanetkar BPB 8
102 Data Communications & Networking Forouzan Mc Graw Hill 4
103 Operating System Galvin Wiley 6
```