ABSTRACT CLASSES AND METHODS

Abstract class

A class that is declared as abstract is known as **abstract class**. It can have abstract and non-abstract methods (method with body). It needs to be extended and its method implemented. It cannot be instantiated.

Syntax:

```
abstract class classname
{
}
```

Abstract method

A method that is declared as abstract and does not have implementation is known as abstract method. The method body will be defined by its subclass.

Abstract method can never be final and static. Any class that extends an abstract class must implement all the abstract methods declared by the super class.

Note:

A normal class (non-abstract class) cannot have abstract methods.

Syntax:

abstract returntype functionname (); //No definition

Syntax for abstract class and method:

```
modifier abstract class className
 {
  //declare fields
  //declare methods
  abstract dataType methodName();
}
modifier class childClass extends className
{
dataType methodName()
{
}
}
```

Example 1

```
//abstract parent class
abstract class Animal
{
  //abstract method
  public abstract void sound();
}
//Lion class extends Animal class
public class Lion extends Animal
{
  public void sound()
{
    System.out.println(“Roars”);
  }
  public static void main(String args[])
{
    Animal obj = new Lion();
    obj.sound();
  }
}
```

Output:

```
Roars
```

In the above code, Animal is an abstract class and Lion is a concrete class.

Example 2

```
abstract class Bank
{
abstract int getRateOfInterest();
}
class SBI extends Bank
{
int getRateOfInterest()
{
   return 7;
}
}
class PNB extends Bank
{
```

```
int getRateOfInterest()
{
   return 8;
}
}
public class TestBank
{
public static void main(String args[])
{
Bank b=new SBI();//if object is PNB, method of PNB will be invoked
int interest=b.getRateOfInterest();
System.out.println("Rate of Interest is: "+interest+" %");
b=new PNB();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
}
}
```

Output:

```
Rate of Interest is: 7 %
Rate of Interest is: 8 %
```

Abstract class with concrete (normal) method

Abstract classes can also have normal methods with definitions, along with abstract methods.

Sample Code:

```
abstract class A
{
 abstract void callme();
 public void normal()
 {
  System.out.println("this is a normal (concrete) method.");
 }
}
public class B extends A
{
 void callme()
 {
  System.out.println("this is an callme (abstract) method.");
 }
 public static void main(String[] args)
 {
  B b = new B();
  b.callme();
  b.normal();
 }
}
```

Output:

```
this is an callme (abstract) method.
this is a normal (concrete) method.
```

Observations about abstract classes in Java

1. **An instance of an abstract class cannot be created; But, we can have referencesof abstract class type though.**

Sample Code:

```
abstract class Base
{
   abstract void fun();
}
class Derived extends Base
{
   void fun()
{
System.out.println("Derived fun() called");
}
}
public class Main
{
   public static void main(String args[])
{
      // Base b = new Base(); Will lead to error
      // We can have references of Base type.
      Base b = new Derived();
```

```
        b.fun();
    }
}
```

Output:

```
Derived fun() called
```

2. **An abstract class can contain constructors in Java. And a constructor of abstract class is called when an instance of a inherited class is created.**

Sample Code:

```
abstract class Base
{
    Base()
    {
        System.out.println("Within Base Constructor");
    }
    abstract void fun();
}
class Derived extends Base
{
    Derived()
    {
        System.out.println("Within Derived Constructor");
    }
    void fun()
    {
        System.out.println(" Within Derived fun()");
    }
}
public class Main
{
    public static void main(String args[])
    {
       Derived d = new Derived();
    }
}
```

Output:

```
Within Base Constructor
Within Derived Constructor
```

3. **We can have an abstract class without any abstract method. This allows us to create classes that cannot be instantiated, but can only be inherited.**

Sample Code:

```
abstract class Base
{
   void fun()
   {
      System.out.println("Within Base fun()");
   }
}
class Derived extends Base
{
}
public class Main
{
   public static void main(String args[])
   {
      Derived d = new Derived();
      d.fun();
   }
}
```

Output:

```
Within Base fun()
```

4. **Abstract classes can also have final methods (methods that cannot be overridden).**

Sample Code:

```
abstract class Base
{
   final void fun()
   {
        System.out.println("Within Derived fun()");
    }
}
class Derived extends Base
{
}
public class Main
{
    public static void main(String args[])
    {
        Base b = new Derived();
        b.fun();
    }
}
```

Output:

```
Within Derived fun()
```

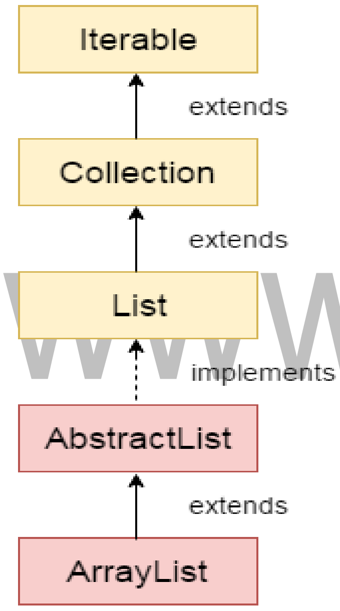
ARRAYLIST

ArrayList is **a part of collection framework**. It is present in **java.util package**. It provides us dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpfulin programs where lots of manipulation in the array is needed.

- ArrayList inherits AbstractList class and implements List interface.
- ArrayList is initialized by a size; however the size can increase if collection grows or shrink if objects are removed from the collection.
- Java ArrayList allows us to randomly access the list.
- ArrayList cannot be used for primitive types, like int, char, etc.
- ArrayList in Java is much similar to vector in C++.

Java ArrayList class

Java ArrayList class extends AbstractList class which implements List interface. The List interface extends Collection and Iterable interfaces in hierarchical order.



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

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.
void trimToSize()	It is used to trim the capacity of this ArrayList instance to be the list's current size.

```
import java.util.*;
class Arraylist_example{
 public static void main(String args[]){
  ArrayList<String> a1=new ArrayList<String>();
  a1.add("Bala");
  a1.add("Mala");
  a1.add("Vijay");
  ArrayList<String> a2=new ArrayList<String>();
  a2.add("kala");
  a2.add("Banu");
  a1.addAll(a2);
  Iterator itr=a1.iterator();
  while(itr.hasNext()){
  System.out.println(itr.next());
  }
 }
}
```

FINAL METHODS AND CLASSES

The final keyword in java is used to restrict the user. The java final keyword can be applied to:

- variable
- method
- class

Java final variable	-	To prevent constant variables
Java final method	-	To prevent method overriding
Java final class	-	To prevent inheritance

Sample Code:

A final variable speed limit is defined within a class Vehicle. When we try to change the value of this variable, we get an error. This is due to the fact that the value of final variable cannot be changed, once a value is assigned to it.

```
public class Vehicle
{
 final int speedlimit=60;//final variable
 void run()
 {
  speedlimit=400;
 }
 public static void main(String args[])
 {
 Vehicle obj=new Vehicle();
 obj.run();
 }
  }
```

Output:

```
/Vehicle.java:6: error: cannot assign a value to final variable speedlimit
  speedlimit=400;
  ^
1 error
```

Blank final variable

A final variable that is not initialized at the time of declaration is known as blank final variable. We must initialize the blank final variable in constructor of the class otherwise it will throw a compilation error.

Sample Code:

```
public class Vehicle
{
 final int speedlimit; //blank final variable
 void run()
 {
 }
 public static void main(String args[])
 {
 Vehicle obj=new Vehicle();
 obj.run();
 }
  }
```

Output:

```
/Vehicle.java:3: error: variable speedlimit not initialized in the default constructor
 final int speedlimit; //blank final variable
           ^
1 error
```

Java Final Method

A Java method with the final keyword is called a final method and it cannot be overridden in the subclass.

In general, final methods are faster than non-final methods because they are not requiredto be resolved during run-time and they are bonded at compile time.

Sample Code:

```
class XYZ
{
  final void demo()
  {
    System.out.println("XYZ Class Method");
  }
```

```
public class ABC extends XYZ
{
  void demo()
  {
    System.out.println("ABC Class Method");
  }
  public static void main(String args[])
  {
    ABC obj= new ABC();
    obj.demo();
  }
}
```

Output:

```
/ABC.java:11: error: demo() in ABC cannot override demo() in XYZ
  void demo()
      ^
  overridden method is final
1 error
```

The following code will run fine as the final method demo() is not overridden. This showsthat final methods are inherited but they cannot be overridden.

Sample Code:

```
class XYZ
{
  final void demo()
  {
    System.out.println("XYZ Class Method");
  }
}
public class ABC extends XYZ
{
  public static void main(String args[])
  {
    ABC obj= new ABC();
    obj.demo();
    }
```

Output:

```
    XYZ Class Method
```

Points to be remembered while using final methods:

- Private methods of the superclass are automatically considered to be final.
- Since the compiler knows that final methods cannot be overridden by a subclass, so these methods can sometimes provide performance enhancement by removing calls to final methods and replacing them with the expanded code of their declarations at each method call location.
- Methods made inline should be small and contain only few lines of code. If it growsin size, the execution time benefits become a very costly affair.
- A final's method declaration can never change, so all subclasses use the same method implementation and call to one can be resolved at compile time. This is knownas *static binding.*

Java Final Class

- Final class is a class that cannot be extended i.e. it cannot be inherited.
- A final class can be a subclass but not a superclass.
- Declaring a class as final implicitly declares all of its methods as final.
- It is illegal to declare a class as both abstract and final since an abstract class is incompleteby itself and relies upon its subclasses to provide complete implementations.
- Several classes in Java are final e.g. String, Integer, and other wrapper classes.
- The final keyword can be placed either before or after the access specifier.

Syntax:

```
final public class A
{
   //code
}
```

```
public final class A
{
   //code
}
```

Sample Code:

```
    final class XYZ
    {
    }

    public class ABC extends XYZ
    {
```

```
    void demo()
    {
      System.out.println("My Method");
    }
    public static void main(String args[])
    {
      ABC obj= new ABC();
      obj.demo();
    }
  }
```

Output:

```
/ABC.java:5: error: cannot inherit from final XYZ
public class ABC extends XYZ
                 ^
1 error
```

Important points on final in Java

- Final keyword can be applied to a member variable, local variable, method or classin Java.
- Final member variable must be initialized at the time of declaration or inside the constructor, failure to do so will result in compilation error.
- We cannot reassign value to a final variable in Java.
- The local final variable must be initialized during declaration.
- A final method cannot be overridden in Java.
- A final class cannot be inheritable in Java.
- Final is a different than finally keyword which is used to Exception handling in Java.
- Final should not be confused with finalize() method which is declared in Object class and called before an object is a garbage collected by JVM.
- All variable declared inside Java interface are implicitly final.
- Final and abstract are two opposite keyword and a final class cannot be abstract in Java.
- Final methods are bonded during compile time also called static binding.
- Final variables which are not initialized during declaration are called blank final variable and must be initialized in all constructor either explicitly or by calling this(). Failure to do so compiler will complain as "final variable (name) might not be initialized".

- Making a class, method or variable final in Java helps to improve performance because JVM gets an opportunity to make assumption and optimization.

INHERITANCE

- Inheritance is the mechanism in java by which **one class is allow to inherit the features of another class.**
- It is process of deriving a new class from an existing class.
- A class that is inherited is called **a *superclass*** and the class that does the inheriting is called a ***subclass.***
- Inheritance represents the IS-A relationship, also known as ***parent child relationship.*** The keyword used for inheritance is **extends.**

Syntax:

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

Here, the extends keyword indicates that we are creating a new class that derives from an existing class.

Note: The constructors of the superclass are never inherited by the subclass

Advantages of Inheritance:

- **Code reusability** - public methods of base class can be reused in derived classes
- **Data hiding** – private data of base class cannot be altered by derived class
- **Overriding**--With inheritance, we will be able to override the methods of the base class in the derived class

Example:

```
// Create a superclass.
class BaseClass
{
   int a=10,b=20;
   public void add()

   {
      System.out.println("Sum:"+(a+b));

      }
   }

public class Main extends BaseClass
{
   public void sub()

   {
    System.out.println("Difference:"+(a-b));
      public static void main(String[] args)
      {

      Main obj=new Main();
      obj.add();
         obj.sub();

      }

 }
```

Sample Output:

```
Sum:30
Difference:-10
```

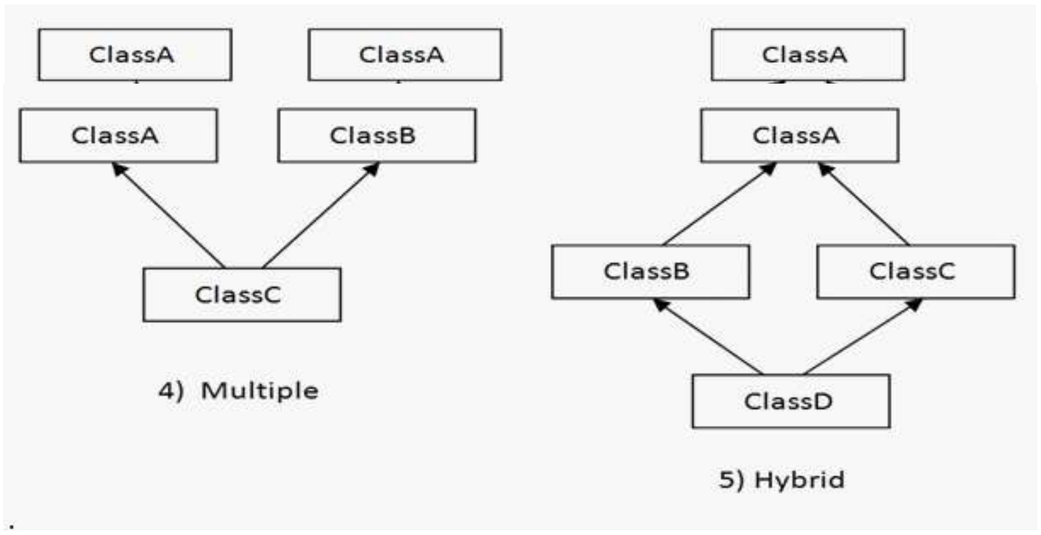
Types of inheritance

Single Inheritance :

In single inheritance, **a subclass inherit the features of one superclass**.

Example:

```
class Shape{

   int a=10,b=20;

}

class Rectangle extends Shape{
   public void rectArea(){
      System.out.println("Rectangle Area:"+(a*b));

public class Main

{

     public static void main(String[] args) {
        Rectangle obj=new Rectangle();
              obj.rectArea();

     }}
```



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 i.e. a derived class in turn acts as a base class for another class.

Example:

```
class Numbers{
   int a=10,b=20;
}
class Add2 extends Numbers{
   int c=30;
   public void sum2(){
      System.out.println(“Sum of 2 nos.:”+(a+b));
   }
}
class Add3 extends Add2{
   public void sum3(){
      System.out.println(“Sum of 3 nos.:”+(a+b+c));
   }
}
public class Main
{
     public static void main(String[] args) {
        Add3 obj=new Add3();
              obj.sum2();
              obj.sum3();
     }
}
```

Sample Output: Sum

```
of 2 nos.:30Sum of
3 nos.:60
```

Hierarchical Inheritance:

In Hierarchical Inheritance, **one class serves as a superclass** (base class) for more than one sub class.

Example:

```
class Shape{
   int a=10,b=20;
}
class Rectangle extends Shape{
   public void rectArea(){
      System.out.println("Rectangle Area:"+(a*b));
   }
}
class Triangle extends Shape{
   public void triArea(){
      System.out.println("Triangle Area:"+(0.5*a*b));
   }
}
public class Main
{
     public static void main(String[] args) {
        Rectangle obj=new Rectangle();
              obj.rectArea();
        Triangle obj1=new Triangle();
              obj1.triArea();
     }
}
```

Sample Output: Rectangle

Area:200Triangle

Area:100.0

Multiple inheritance

Java does not allow multiple inheritance:

- **To reduce the complexity and simplify the language**
- ***To avoid the ambiguity caused by multiple inheritance***

For example, Consider a class C derived from two base classes A and B. Class C inheritsA and B features. If A and B have a method with same signature, there will be ambiguity to call method of A or B class. It will result in compile time error.

```
class A{
void msg(){System.out.println("Class A");}
}
class B{
```

```
void msg(){System.out.println("Class B ");}
}
class C extends A,B{//suppose if it were
  Public Static void main(String args[]){C
  obj=new C();
  obj.msg();//Now which msg() method would be invoked?
}
}
```

Sample Output:

```
Compile time error
```

Direct implementation of multiple inheritance is not allowed in Java. But it is achievable using Interfaces. The concept about interface is discussed in chapter.2.7.

Access Control in Inheritance

The following rules for inherited methods are enforced −

- Variables declared public or protected in a superclass are inheritable in subclasses.
- Variables or Methods declared private in a superclass are not inherited at all.
- Methods declared public in a superclass also must be public in all subclasses.
- Methods declared protected in a superclass must either be protected or public in subclasses; they cannot be private.

Example:

```
// Create a superclass
class A{
   int x;              // default specifier
   private int y;      // private to A

   public void set_xy(int a,int b){
      x=a;
      y=b;
   }
}
// A's y is not accessible here.
class B extends A{
   public void add(){
      System.out.println("Sum:"+(x+y)); //Error: y has private access in A – not inheritable
   }
}
class Main{
    public static void main(String args[]){B
        obj=new B();
        obj.set_xy(10,20);
        obj.add();
    }
}
```

In this example since y is declared as private, it is only accessible by its own class members. Subclasses have no access to it.

INTERFACES

An interface is a reference type in Java. It is similar to class. It is a **collection of abstract methods**. Along with abstract methods, an interface may **also contain constants, default methods, static methods, and nested types**. Method bodies exist only for default methods and staticmethods.

An interface is similar to a class in the following ways:

- An interface **can contain any number of methods**.
- **An interface is written in a file with a .java extension**, with the name of the interfacematching the name of the file.
- **The byte code of an interface appears in a .class file.**
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

Uses of interface:

- Since java does not support **multiple inheritance** in case of class, it can be achievedby using interface.
- It is also used to achieve **loose coupling.**
- Interfaces are used to implement **abstraction**.

Defining an Interface

An interface is defined much like a class.

Syntax:

accessspecifier interface interfacename

{

return-type method-name1(parameter- list);

return-type method-name2(parameter-list);

type final-varname1 = value;

type final-varname2 =value;

// ...

return-type method-nameN(parameter-list);

type final-varnameN = value;

}

When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as public, the interface can be used by any other code.

- The java file must have the same name as the interface.
- The methods that are declared have no bodies. They end with a semicolon after the

parameter list. They are abstract methods; there can be no default implementation of any method specified within an interface.

- Each class that includes an interface must implement all of the methods.
- Variables can be declared inside of interface declarations. They are implicitly final and static, meaning they cannot be changed by the implementing class. They must also be initialized.
- All methods and variables are implicitly public.

Sample Code:

The following code declares a simple interface Animal that contains two methods called eat() and travel() that take no parameter.

```
/* File name : Animal.java
*/interface Animal {
  public void eat();
  public void travel();
}
```

Implementing an Interface

Once an interface has been defined, one or more classes can implement that interface. To implement an interface, the **'implements'** clause is included in a class definition and then the methods defined by the interface are created.

Syntax:

```
class classname [extends superclass] [implements interface [,interface...]]
{
// class-body
}
```

Properties of java interface

- If a class implements more than one interface, the interfaces are separated with a comma.
- If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface.
- The methods that implement an interface must be declared public.
- The type signature of the implementing method must match exactly the type signature specified in the interface definition.

Rules

- A class can implement more than one interface at a time.
- A class can extend only one class, but can implement many interfaces.
- An interface can extend another interface, in a similar way as a class can extend another class.

Sample Code 1:

The following code implements an interface Animal shown earlier.

```
/* File name : MammalInt.java */
public class Mammal implements Animal
{
  public void eat()
{
    System.out.println("Mammal eats");
  }
  public void travel()
{
    System.out.println("Mammal travels");
  }
  public int noOfLegs()
{
    return 0;
  }
  public static void main(String args[])
{
    Mammal m = new Mammal();
    m.eat();
    m.travel();
  }
}
```

Output:

```
Mammal eats Mammal
travels
```

It is both permissible and common for classes that implement interfaces to define additional members of their own. In the above code, Mammal class defines additional method called noOfLegs().

Sample Code 2:

The following code initially defines an interface 'Sample' with two members. This interface is implemented by a class named 'testClass'.

```
import java.io.*;
// A simple interface
interface Sample
```

```
{
   final String name = “Shree”;
   void display();
}
 // A class that implements interface.
public class testClass implements Sample
{
   public void display()
   {
      System.out.println(“Welcome”);
   }
   public static void main (String[] args)
   {
      testClass t = new testClass();
      t.display();
      System.out.println(name);
   }
}
```

Output:

```
Welcome
Shree
```

Sample Code 3:

In this example, Drawable interface has only one method. Its implementation is providedby Rectangle and Circle classes.

```
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”);
}
}
public class TestInterface
{
public static void main(String args[])
{
Drawable d=new Circle();
d.draw();
}
  }
```

Output:

```
Drawing circle
```

Nested Interface

An interface can be declared as a member of a class or another interface. Such an interface is called a member interface or a nested interface. A nested interface can be declared as public, private, or protected.

Sample Code:

```
interface MyInterfaceA
{
   void display(); interface
   MyInterfaceB
   {
      void myMethod();
   }
}
public class NestedInterfaceDemo1 implements MyInterfaceA.MyInterfaceB
{
    public void myMethod()
   {
       System.out.println(“Nested interface method”);
    }
    public static void main(String args[])
   {
       MyInterfaceA.MyInterfaceB obj= new NestedInterfaceDemo1();
    obj.myMethod();
    }
}
```

Output:

```
Nested interface method
```

JAVA STRING

In general **string is a sequence of characters**. String is an object that represents a sequenceof characters. The **java.lang.String class is used to create string object**. In java, string is basically an object that represents sequence of char values. An array of characters works same asjava string. For example:

Java String class provides a lot of methods to perform operations on string such as com- pare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence* in-terfaces. The CharSequence interface is used to represent sequence of characters. It is imple-mented by String, StringBuffer and StringBuilder classes. It means can create string in java by using these 3 classes.

The string objects can be created using two ways.

1. By String literal
2. By new Keyword

String Literal

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

1. String s=”welcome”;

Each time you create a string literal, the JVM checks the string constant pool first. If the string already exists in the pool, a reference to the pooled instance is returned. If string doesn’t exist in the pool, a new string instance is created and placed in the pool. For example:

String s1=”Welcome”;

String s2=”Welcome”;

In the above example only one object will be created. Firstly JVM will not find any string object with the value “Welcome” in string constant pool, so it will create a new object. Afterthat it will find the string with the value “Welcome” in the pool, it will not create new object but will return the reference to the same instance. To make Java more memory efficient (be-cause no new objects are created if it exists already in string constant pool).

2. By new keyword

String s=new String(“Welcome”);

In such case, JVM will create a new string object in normal (non pool) heap memory and the literal “Welcome” will be placed in the string constant pool. The variable s will refer to the object in heap (non pool).

The java String is immutable i.e. it cannot be changed. Whenever we change any string, a new instance is created. For mutable string, you can use StringBuffer and StringBuilder classes.

The following program explains the creation of strings

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

{

String s1=”java”;
char c[]={‘s’,’t’,’r’,’i’,’n’,’g’};String s2=new String(c);
String s3=new String(“example”);
System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
}}
```

Java String class methods

The java.lang.String class provides many useful methods to perform operations on se- quence of char values.

Method	**Descriptio n**
char charAt(int index)	returns char value for the particular index
int length()	returns string length
static String format(String format,Object... args)	returns formatted string
static String format(Locale l, Stringformat, Object... args)	returns formatted string with given locale
String substring(int beginIndex)	returns substring for given begin index
String substring(int beginIndex, intendIndex)	returns substring for given begin index andend index
boolean contains(CharSequence s)	returns true or false after matching the se- quence of char value
static String join(CharSequence delim- iter, CharSequence... elements)	returns a joined string
boolean equals(Object another)	checks the equality of string with object
boolean isEmpty()	checks if string is empty
String concat(String str)	concatinates specified string
String replace(char old, char new)	replaces all occurrences of specified char value

String replace(CharSequence old, Char- Sequence new)	replaces all occurrences of specified CharSe- quence
static String equalsIgnoreCase(String another)	compares another string. It doesn't check case.
String[] split(String regex)	returns splitted string matching regex
String[] split(String regex, int limit)	returns splitted string matching regex and limit
String intern()	returns interned string
int indexOf(int ch)	returns specified char value index
int indexOf(int ch, int fromIndex)	returns specified char value index starting with given index
int indexOf(String substring)	returns specified substring index
int indexOf(String substring, int fro- mIndex)	returns specified substring index starting with given index
String toLowerCase()	returns string in lowercase.
String toLowerCase(Locale l)	returns string in lowercase using specified locale.
String toUpperCase()	returns string in uppercase.
String toUpperCase(Locale l)	returns string in uppercase using specified locale.
String trim()	removes beginning and ending spaces of this string.
static String valueOf(int value)	converts given type into string. It is over- loaded

The following program is an example for String concat function:

```
class string_method{
 public static void main(String args[]){
  String s=”Java”;
  s=s.concat(“ Programming”);
  System.out.println(s);
 }
}
```

Output:

```
Java Programming
```

NESTED CLASSES

In Java, **a class can have another class as its member**. The class written within another class is called the nested class, and the class that holds the inner class is called the outer class.

Java inner class is defined inside the body of another class. Java inner class can be declared private, public, protected, or with default access whereas an outer class can have only public or default access.

The syntax of nested class is shown below:

```
class Outer_Demo

{

class Nested_Demo

{
      }
    }
```

Types of Nested classes

There are two types of nested classes in java. They are non-static and static nested classes. The non-static nested classes are also known as inner classes.

- Non-static nested class (inner class)
  - Member inner class
  - Method Local inner class
  - Anonymous inner class
- Static nested class

Type	Description
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing interface or extending class. Its name is decided by the java compiler.
Method Local Inner Class	A class created within method.
Static Nested Class	A static class created within class.
Nested Interface	An interface created within class or interface.

INNER CLASSES (NON-STATIC NESTED CLASSES)

Inner classes can be used as the security mechanism in Java. Normally, a class cannot be related with the access specifier **private.** However if a class is defined as a member of other class, then the inner class can be made private. This class can have access to the private members of a class.

The three types of inner classes are

- Member Inner Class
- Method-local Inner Class
- Anonymous Inner Class

Member Inner Class

The **Member inner class is a class written within another class**. Unlike a class, an inner class can be private and once you declare an inner class private, it cannot be accessed from an object outside the class.

The following program is an example for member inner class.

```
 class Outer_class {
   int n=20;
   private class Inner_class {
   public void display() {
   System.out.println("This is an inner class");
   System.out.println("n:"+n);
     }
   }
     void print_inner() {
      Inner_class inn = new Inner_class();
      inn.display();
   }
 }
   public class Myclass {
   public static void main(String args[]) {
     Outer_class out= new Outer_class();
     out.print_inner();
   }
 }
```

Output:

```
This is an inner class
```

Method-local Inner Class

In Java, a class can be written within a method. Like local variables of the method, the scope of the inner class is restricted within the method. A method-local inner class can be instantiated only within the method where the inner class is defined. The following program shows how to use a method-local inner class. The following program is an example for Method-local Inner Class

```
public  class  Outer_class  {
    void Method1() {
        int n = 100;
       class   MethodInner_class   {
       public void display() {
          System.out.println(“This is method inner class “);
          System.out.println(“n:”+n);
        }
      }
      MethodInner_class inn= new MethodInner_class();
      inn.display();
    }
    public static void main(String args[]) {
      Outer_class out = new Outer_class();
      out.Method1();
    }
  }
```

Output:

```
This is method inner class
n: 100
```

Anonymous Inner Class

An inner class declared without a class name is known as an anonymous inner class. The anonymous inner classes can be created and instantiated at the same time. Generally, they are used whenever you need to override the method of a class or an interface. The syntax of an anonymous inner class is as follows –

```
abstract class Anonymous_Inner {
  public abstract void Method1();
}
```

The following program is an example for anonymous inner class.

```
public class Outer_class {
  public static void main(String args[]) {
    Anonymous_Inner inn = new Anonymous_Inner() {
      public void Method1() {
         System.out.println(“This is the anonymous inner class”);
      }
    };
    inn.Method1();
  }
}
```

Output:

```
This is the anonymous inner class
```

Static Nested Class

A static inner class is a nested class which is a static member of the outer class. It canbe accessed without instantiating the outer class, using other static members. Just like static members, a static nested class does not have access to the instance variables and methods of the outer class. Instantiating a static nested class is different from instantiating an inner class. The following program shows how to use a static nested class.

```
public  class  Outer_class  {
  static class inner_class{
    public void Method1() {
    System.out.println("This is the nested class");
    }
  }
    public static void main(String args[]) {
    Outer_class.inner_class obj = new Outer_class.inner_class();
    obj.Method1();
  }
}
```

Output:

```
This is the nested class
```

Advantage of java inner classes:

There are basically three advantages of inner classes in java. They are as follows:

- Nested classes represent a special type of relationship that is it can access all the members of outer class including private.
- Nested classes are used to develop more readable and maintainable code because it logically group classes and interfaces in one place only.
- It provides code optimization. That is it requires less code to write.

2.1 OBJECT CLONING

Object cloning refers to creation of **exact copy of an object**. It creates a new instance of the class of current object and initializes all its fields with exactly the contents of the corresponding fields of this object. **In Java, there is no operator to create copy of an object.** Unlike C++, in Java, if we use assignment operator then it will create a copy of reference variable and not the object. This can be explained by taking an example. Following program demonstrates the same.

// Java program to demonstrate that assignment operator creates a new reference to same object.

```
import
java.io.*;class
sample
{
    int a;
    float b;
    sample()
    {
            a = 10;
            b = 20;
    }
}
class Mainclass
{
    public static void main(String[] args)
    {
            sample ob1 = new sample();
            System.out.println(ob1.a + " " +
            ob1.b);sample ob2 = ob1;
            ob2.a = 100;
            System.out.println(ob1.a+"
            "+ob1.b);System.out.println(ob2.a+"
            "+ob2.b);
    }
}
```

Output:

```
10 20.0
100 20.0
100 20.0
```

Creating a copy using clone() method

The class whose object's copy is to be made must have a public clone method in it or inone of its parent class.

- Every class that implements clone() should call **super.clone()** to obtain the cloned
  object reference.

The class must also implement **java.lang.Cloneable** interface whose object clonewe want to create otherwise it will throw **CloneNotSupportedException** Syntax:

protected Object clone() throws CloneNotSupportedException

```
import  ava.util.ArrayList;

class sample1
{
    int a, b;
}
class sample2 implements Cloneable
{
    int c;int d;
    sample1 s = new sample1();
    public Object clone() throws CloneNotSupportedException
    {
        return super.clone();
    }
}
```

- ***public class***when clone method is called on that class's object.

```
    sample2 ob1 = new
    sample2();ob1.c = 10;
    ob1.d = 20;
    ob1.s.a = 30;
    ob1.s.b = 40;
    sample2 ob2 = (sample2)ob1.clone();
    ob2.d = 100; //Change in primitive type of ob2 will not be reflected in ob1 field
```

```
        ob2.s.a = 300; //Change in object type  field  will  be  reflected  in both ob2
        and ob1(shallow copy)
        System.out.println(ob1.c + " " + ob1.d + " " +ob1.s.a + " " + ob1.s.b);
        System.out.println(ob2.c + " " + ob2.d + " " +ob2.s.a + " " + ob2.s.b);
        }
    }
```

Types of Object cloning

1. Deep Copy
2. Shallow Copy

Shallow copy

Shallow copy is method of copying an object. It is the default in cloning. In this method the **fields of an old object ob1 are copied to the new object ob2**. While copying the object type field the reference is copied to ob2 i.e. object ob2 will point to same location as pointed out by ob1. If the field value is a primitive type it copies the value of the primitive type. So, any changes made in referenced objects will be reflected in other object.

Note:

Shallow copies are cheap and simple to make.

Deep Copy

To create **a deep copy of object ob1 and place it in a new object ob2 then new copy of any referenced objects fields are created and these references are placed in object ob2.** This means any changes made in referenced object fields in object ob1 or ob2 will be reflected only in that object and not in the other. A deep copy copies all fields, and makes copies of dynamically allocated memory pointed to by the fields. A deep copy occurs when an object is copied along with the objects to which it refers.

```
//Java   program   for   deep   copy   using
clone()

import java.util.ArrayList;
    class Test
    {
        int a, b;
    }
    class Test2 implements Cloneable
    {
        int c, d;
        Test ob1 = new Test();
        public Object clone() throws CloneNotSupportedException
        {
```

```
                    // Assign the shallow copy to new refernce variable t Test2
                    t1 = (Test2)super.clone();
}
public class Main
{
   public static void main(String args[]) throws CloneNotSupportedException
   {
   Test2 t2 = new Test2(); t2.c = 10;
   t2.d = 20;
   t2.ob1.a = 30;
   t2.ob1.b = 40; Test2 t3 =
   (Test2)t2.clone();t3.c = 100;
   t3.ob1.a = 300;
   System.out.println (t2.c + " " + t2.d + " " + t2.ob1.a + " " + t2.ob1.b);
   System.out.println (t3.c + " " + t3.d + " " + t3.ob1.a + " " + t3.ob1.b);
}  }
                    t1.ob1 = new Test();
                    // Create a new object for the field c
                    // and assign it to shallow copy obtained,
                    // to make it a deep copy
                    return t1;
      }
```

Output

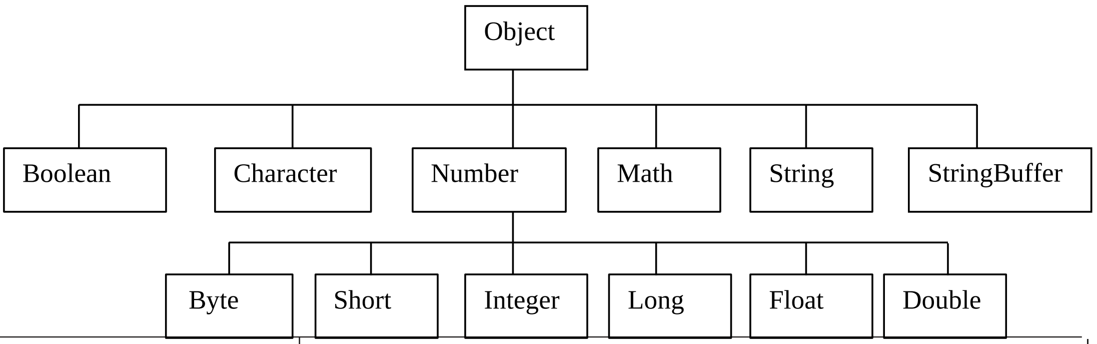
```
10 20 30 40
    100 20 300 0
```

Advantages of clone method:

- If we use assignment operator to assign an object reference to another reference variable then it will point to same address location of the old object and no new copyof the object will be created. Due to this any changes in reference variable will be reflected in original object.
- If we use copy constructor, then we have to copy all of the data over explicitly i.e. we have to reassign all the fields of the class in constructor explicitly. But in clone method this work of creating a new copy is done by the method itself. So to avoid extra processing we use object cloning.

THE OBJECT CLASS

The Object class **is the parent class of all the classes in java by default** (directly or indirectly). The java.lang.Object class is the root of the class hierarchy. Some of the Object class are Boolean, Math, Number, String etc.



Object class Methods	**Description**
boolean equals(Object)	Returns true if two references point to the same object.
String toString()	Converts object to String
void notify() void notifyAll() void wait()	Used in synchronizing threads
void finalize()	Called just before an object is garbage collected
Object clone()	Returns a new object that are exactly the same as the current object
int hashCode()	Returns a hash code value for the object.

Example:

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

                                          /*hashcode is the unique number generated by JVM*/
      Test t = new Test(); System.out.println(t);

      System.out.println(t.toString());      // provides String representation of an Object
      System.out.println(t.hashCode());

      System.out.println("end");
   }
   protected void finalize() // finalize() is called just once on an object
   {
      System.out.println("finalize method called");
   }
```

Sample Output:

```
Test@2a139a55
Test@2a139a55
705927765
```

USING SUPER

The super keyword **refers to immediate parent class object**. Whenever you create the in- stance of subclass, an instance of parent class is created implicitly which is referred by superreference variable.

- It an be used to refer immediate parent class instance variable when both parent and child class have member with same name
- It can be used to invoke immediate parent class method when child class has overridden that method.
- super() can be used to invoke immediate parent class constructor.

Use of super with variables:

When both parent and child class have member with same name, we can use super key-word to access member of parent class.

Example:

```
class SuperCls
{
   int x = 20;
}
 /* sub class SubCls extending SuperCls */
class SubCls extends SuperCls
{
   int x = 80;
    void display()
   {
      System.out.println("Super Class x: " + super.x); //print x of super class
      System.out.println("Sub Class x: " + x); //print x of subclass
   }
}
 /* Driver program to test
*/class Main
{
   public static void main(String[] args)
   {
      SubCls obj = new
      SubCls();obj.display();
   }
}
```

Sample Output:

```
Super Class x:
20 Sub Class
x: 80
```

In the above example, both base class and subclass have a member x. We could access x of base class in sublcass using super keyword.

Use of super with methods:

The super keyword can also be used to invoke parent class method. It should be used ifsubclass contains the same method as parent class (Method Overriding).

```
class SuperCls
{
    int x = 20;
    void display(){ //display() in super class
        System.out.println("Super Class x: " + x);
    }
}
 /* sub class SubCls extending SuperCls */
class SubCls extends SuperCls
{
    int x = 80;
     void display()    //display() redefined in sub class – method overriding
    {
        System.out.println("Sub Class x: " + x);
        super.display();        // invoke super class
        display()
    }
}
 /* Driver program to test
*/class Main
{
    public static void main(String[] args)
    {
        SubCls obj = new
        SubCls();obj.display();
    }
```

Sample Output:

```
Sub Class x: 80
Super Class x: 20
```

In the above example, if we only call method display() then, the display() of sub class gets invoked. But with the use of super keyword, display() of superclass could also be invoked.

Use of super with constructors:

The super keyword can also be used to invoke the parent class constructor.

Syntax:

```
super();
```

- super() if present, must always be the first statement executed inside a subclass constructor.
- When we invoke a super() statement from within a subclass constructor, we are invoking the immediate super class constructor

Example:

```
class SuperCls
{
   SuperCls(){
      System.out.println("In Super Constructor");
   }
}
/* sub class SubCls extending SuperCls */
class SubCls extends SuperCls
{
      SubCls(){
        super();
      System.out.println("In Sub Constructor");
   }
}
/* Driver program to test */
class Main
{
   public static void main(String[] args)
   {
      SubCls obj = new SubCls();
   }
}
```

Sample Output:

```
In Super ConstructorIn
```

Sub Constructor

Invoking Superclass Parameterized Constructor

To call parameterized constructor of superclass, we must use the super keyword as shown below.

Syntax:

```
super(value);
```

Example:

```
class SuperCls{ int
   x; SuperCls(int
   x){
      this.x=x;                // this refers to current invoking object
   }
}
class SubCls extends SuperCls{
   int y;
   SubCls(int x,int y){
      super(x);                // invoking parameterized constructor of superclass
      this.y=y;
   }
   public void display(){ System.out.println("x:
      "+x+" y: "+y);
   }
}
public class Main
{
     public static void main(String[] args) {
        SubCls obj=new SubCls(10,20);
        obj.display();
     }
}
```

Sample Output:

```
x: 10 y: 20
```

The program contains a superclass and a subclass, where the superclass contains a parameterized constructor which accepts a integer value, and we used the super keyword to invokethe parameterized constructor of the superclass.