Strings:

In java, string is basically an object that represents sequence of char values. An array of characters works same as java string.

String is a class in *java.lang* package. But in java, all classes are also considered as data types. So, we can take string as a data type also. A class is also called as user-defined data type.

Creating Strings:

There are three ways to create strings in Java:

- We can create a string just by assigning a group of characters to a string type variable:

```
String s;
s = "Hello";
```

- Preceding two statements can be combined and written as:

```
String s = "Hello";
```

In this, case JVM creates an object and stores the string: "Hello" in that object. This object is referenced by the variable's'. Remember, creating object means allotting memory for storing data.

- We can create an object to String class by allocating memory using new operator. This is just like creating an object to any class, like given here:

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

  Here, we are doing two things. First, we are creating object using new operator. Then, we are storing the string: "Hello" into the object.

- The third way of creating the strings is by converting the character arrays into strings. Let us take a character type array: arr[] With some characters, as:

```
char arr[]={'H','e','l','l','o'};
```

- Now create a string object, by pass.ing the array name to it, as:

```
String s = new String(arr);
```

String Methods:

No.	Method	Description
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	`String substring(int beginIndex, int endIndex)`	returns substring for given begin index and end index
5	`boolean contains(CharSequence s)`	returns true or false after matching the sequence of char value
6	`static String join(CharSequence delimiter, CharSequence... elements)`	returns a joined string
7	`boolean equals(Object another)`	checks the equality of string with object
8	`boolean isEmpty()`	checks if string is empty
9	`String concat(String str)`	concatenates specified string
10	`String replace(char old, char new)`	replaces all occurrences of specified char value
11	`static String equalsIgnoreCase(String another)`	Compares another string. It doesn't

		check case.
12	`String[] split(String regex, int limit)`	returns splitted string matching regex and limit
13	`int indexOf(int ch)`	returns specified char value index
14	`String toLowerCase()`	Returns string in lowercase.
15	`String toUpperCase()`	Returns string in uppercase.
16	`String trim()`	Removes beginning and ending spaces of this string.
17	`static String valueOf(int value)`	Converts given type into string. It is overloaded.

equals method in java:

Program:

```
class StringComapre
{
	public static void main(String args[])
	{
		String s1 = "Hello";
		String s2 = new String("Hello");
		System.out.println(s1==s2); //false
		System.out.println(s1.equals(s2)); //true
	}
}
```

Immutability of String:

- In java, **string objects are immutable**. Immutable simply means unmodifiable or unchangeable.
- Once string object is created its data or state can't be changed but a new string object is created.

```
class Testimmutablestring{
 public static void main(String args[]){
   String s="Sachin";
   s.concat(" Tendulkar");
   System.out.println(s);
 }
}
```

Output: Sachin

- But if we explicitely assign it to the reference variable, it will refer to "Sachin Tendulkar" object.For example:

```
class Testimmutablestring1{
 public static void main(String args[]){
   String s="Sachin";
   s=s.concat(" Tendulkar");
   System.out.println(s);
 }
}
```

Output: Sachin Tendulkar

Classes and Objects:

We know a class is a model for creating objects. This means, the properties and actions of the objects are written in the class. Properties are represented by variables and actions of the objects are represented by methods. So, a class contains variables and methods. The same variables and methods are also available in the objects because they are created from the class. These variables are also called 'instance variables' because they are created inside the object (instance).

If we take Person class, we can write code in the class that specifies the properties and actions performed by any person. For example, a person has properties like name, age, etc. Similarly a person can perform actions like talking; walking, etc. So, the class Person contains these properties and actions as shown here:

```
class Person
{
	String name;
	int age;
	void talk()
	{
		System.out.println("Hello my name is "+name);
		System.out.println("and my age is "+age);
	}
}
```

Object Creation:

We know that the class code along with method code is stored in 'method area' of the JVM. When an object is created, the memory is allocated on 'heap'. After creation of an object, JVM produces a unique reference number for the object from the memory address of the object. This reference number is also called hash code number.

To know the hashcode number (or reference) of an object, we can use hashCode() method object class, as shown here:

```
Person p = new Person();
System.out.println(p.hashCode());
```

The object reference, (hash code) internally represents heap memory where instance-variables are stored. There would be a pointer (memory address) from heap memory to a special structure located in method area. In method area, a table is available which contains pointers to static variables and methods.

```
class Person
{
	String name;
	int age;
	void talk()
	{
		System.out.println("Hello my name is "+name);
		System.out.println("and my age is "+age);
	}
}
class Demo
{
	public static void main(String args[])
```

```
        {
                Person p = new Person();
                System.out.println(p.hashCode());   // 705927765
        }
}
```

Initializing the instance variables:

It is the duty of the programmer to initialize the instance depending on his requirements. There are various ways to do this.

First way is to initialize the instance variables of Person class in other class, i.e., Demo class. For this purpose, the Demo class can be rewritten like this:

```
class Demo
{
        public static void main(String args[])
        {
                Person p = new Person();
                p.name="Raju";
        p.age=22;
                System.out.println(p.hashCode());   // 705927765
        }
}
```

Second way is to initialize the instance variables of Person class in the Same class. For this purpose, the Person class can be rewritten like this:

```
class Person
{
        String name="Raju";
        int age=22;
        void talk()
        {
                System.out.println("My Name is "+name);
                System.out.println("My Age is "+age);
        }
}
class Demo
{
        public static void main(String args[])
        {
                Person p1 = new Person();
                System.out.println("p1 hashcode "+p1.hashCode());
                p1.talk();
                Person p2 = new Person();
                System.out.println("p2 hashcode "+p2.hashCode());
                p2.talk();
        }
}
```

Output:

```
p1 hashcode 705927765
My Name is Raju
```

```
My Age is 22
p2 hashcode 705934457
My Name is Raju
My Age is 22
```

But, the problem in this way of initialization is that all the objects are initializing with same data.

Constructors:

The third possibility of initialization is using constructors. A constructor is similar to a method is used to initialize the instance variables. The sole purpose of a constructor is to initialize the instance variables. A constructor has the following characteristics:

- The constructor's name and class name should be same. And the constructor's name should end with a pair of simple braces.

```
Person()
{
}
```

- A constructor may have or may not have parameters: Parameters are variables to receive data from outside into the constructor. If a constructor does not have any parameters, it is called 'Default constructor'. If a constructor has 1 or more parameters, it is called 'parameterized constructor'; For example, we can write a default constructor as:

```
Person()
{
}
```

  And a Parameterized constructor with two parameters, as:

```
Person(String s, int i)
{
}
```

- A constructor does not return any value, not even 'void'. Recollect, if a method does not return any value, we write 'void' before the method name. That means, the method is returning 'void' which means 'nothing'. But in case of a constructor, we should not even write 'void' before the constructor.
- A constructor is automatically called and executed at the time of creating an object. While creating an object, if nothing is passed to, the object, the default constructor is called and executed. If-some values are passed to the object, then the parameterized constructor is called. For example, if we create the object as:

```
Person p = new Person(); // Default Constructor
Person p = new Person(Raju, 22); // Parameterized Constructor
```

- A constructor is called and executed only once per object. This means .when we create an object, the constructor is called. When we create second object, again the constructor is called second time.

```
class Person
{
        String name;
        int age;
```

```
        Person()
        {
              name="Raju";
              age=22;
        }
        void talk()
        {
              System.out.println("Hello my name is "+name);
              System.out.println("and my age is "+age);
        }
  }
  class Demo
  {
        public static void main(String args[])
        {
              Person p = new Person(); // Default Constructor
              System.out.println("p hashcode "+p.hashCode());
              p.talk();
              Person p1 = new Person();
              System.out.println(p1.hashCode());
              p1.talk();
        }
  }
```

Output:

```
  p hashcode 705927765
  My Name is Raju
  My Age is 22
  p1 hashcode 705934457
  My Name is Raju
  My Age is 22
```

From the output, we can understand that the same data "Raju" and 22 are store in both objects p1 and p2. p2 object should get p2's data, not p1's data. Isn't it? To mitigate the problem, let us try parameterized constructor, which accepts data from outside and initializes instance variables with that data.

```
  class Person
  {
        String name;
        int age;
        Person()
        {
              name="Raju";
              age=22;
        }
        Person(String s, int i)
        {
              name=s;
              age=i;
```

```
        }
        void talk()
        {
              System.out.println("My Name is "+name);
              System.out.println("My Age is "+age);
        }
  }
  class Demo
  {
        public static void main(String args[])
        {
              Person p1 = new Person();
              System.out.println("p1 hashcode "+p1.hashCode());
              p1.talk();
              Person p2 = new Person("Sita",23);
              System.out.println(p2.hashCode());
              p2.talk();
        }
  }
```

Output:

```
p1 hashcode 705927765
My Name is Raju
My Age is 22
p2 hashcode 705934457
My Name is Sita
My Age is 23
```

Q: When is a constructor called, before or after creating the object?

Ans: *A constructor is called concurrently when the object creation is going on. JVM first allocates memory for the object and then executes the constructor to initialize the instance variables. By the time, object creation is completed; the constructor execution is also completed.*

Difference between Default Constructor and Parameterized Constructor:

Default Constructor	Parameterized Constructor
Default constructor is useful to initialize all the objects with same data.	Parameterized constructor is useful to initialize each object with different data.
Default constructor does not have any parameters.	Parameterized constructor will have 1 or more parameters.
When the data is not passed at the time of creating object, default constructor is called.	When the data is passed at the time of creating object, default constructor is called.

Difference between Constructor and Method:

Java Constructor	Java Method
Constructor is used to initialize the state of an object.	Method is used to expose behaviour of an object.
Constructor must not have return type.	Method must have return type.
Constructor is invoked implicitly.	Method is invoked explicitly.
The java compiler provides a default constructor if you don't have any constructor.	Method is not provided by compiler in any case.
Constructor name must be same as the class name.	Method name may or may not be same as class name.

Q: *What is constructor overloading?*

Ans: *Writing two or more constructors with the same name but with difference in the parameters is called constructor overloading. Such constructors are useful to perform different tasks.*

Write a Java Program for Constructor Overloading?

```
class Box
{
	double width, height, depth;
	Box()
	{
		width = height = depth = 0;
	}
	Box(double len)
	{
		width = height = depth = len;
	}
	Box(double w, double h, double d)
	{
		width = w;
		height = h;
		depth = d;
	}

double volume()
	{
		return width * height * depth;
	}
}
public class Test
{
	public static void main(String args[])
	{
```

```
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mybox3 = new Box(7);
        double vol;
        vol = mybox1.volume();
        System.out.println(" Volume of mybox1 is " + vol);
        vol = mybox2.volume();
        System.out.println(" Volume of mybox2 is " + vol);
        vol = mybox3.volume();
        System.out.println(" Volume of mybox3 is " + vol);
    }
}
```

Output:

```
Volume of mybox1 is 3000.0
Volume of mybox2 is 0.0
Volume of mybox3 is 343.0
```

Java Garbage Collection

- In java, garbage means unreferenced objects. Garbage Collection is process of reclaiming the runtime unused memory automatically.
- In other words, it is a way to destroy the unused objects. To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

Advantage of Garbage Collection

- It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
- It is **automatically done** by the garbage collector (a part of JVM) so we don't need to make extra efforts.

How can an object be unreferenced?

- Even though programmer is not responsible to destroy useless objects but it is highly recommended to make an object unreachable (thus eligible for GC) if it is no longer required.
- There are many ways:
  - By nulling the reference
  - By assigning a reference to another
  - By annonymous object etc.

1) By nulling a reference:

```
Employee e=new Employee();
e=null;
```

2) By assigning a reference to another:

```
Employee e1=new Employee();
Employee e2=new Employee();
e1=e2;//now the first object referred by e1 is available for garbage collection
```

3) By annonymous object:

```
new Employee();
```

finalize() method

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as: *protected void finalize(){}*

gc() method

The *gc()* method is used to invoke the garbage collector to perform cleanup processing. The *gc()* is found in System and Runtime classes.

```
public static void gc(){}
```

Program:

```
public class TestGarbage1
{
   public void finalize()
   {
     System.out.println("object is garbage collected");
   }
   public static void main(String args[])
   {
      TestGarbage1 s1=new TestGarbage1();
      TestGarbage1 s2=new TestGarbage1();
      s1=null;
      s2=null;
      System.gc();
   }
}
```

Output:

```
object is garbage collected
object is garbage collected
```

Methods in Java:

- A method represents a group of statements that performs a task. Here 'task' represents a calculation or processing of data or generating a report etc.
- A Method has two types:
  - Method Header
  - Method Body
- **Method Header:** It contains method name, method parameters and method return data type.

  ```
  returntype methodName(parameters)
  ```
- **Method Body:** Method body consists of group of statements which contains logic to perform the task.

  ```
  {
      Statements;
  }
  ```
- If a method returns some value, then a return statement should be written within the body of method, as:

  ```
  return somevalue;
  ```

- There are two types of methods in java:
  a) Instance methods
  b) Static methods

a) Instance Methods:

Instance methods are the methods which ***act*** on the instance variables of the class. To call the instance methods, we should use the form:

```
objectname. methodname ();
```

Example-1: Write a program to perform addition of two numbers using instance method without return statement.

```
class Addition
{
	void add(double a,double b)
	{
		double c=a+b;
		System.out.println("Addition is "+c);
	}
}
class MethodDemo
{
	public static void main(String args[])
	{
		Addition a1=new Addition();
		a1.add(27,35);
	}
}
```

Output:

```
Addition is 62
```

Example-2: Write a program to perform addition of two numbers using instance method with return statement.

```
class Addition
{
	double add(double a,double b)
	{
		double c=a+b;
		return c;
	}
}
class MethodDemo
{
	public static void main(String args[])
	{
		Addition a1=new Addition();
		double c=a1.add(27,35);
	    System.out.println("Addition is "+c);
	}
}
```

Output:

```
Addition is 62
```

b) Static Methods:

Static methods are the methods which ***do not act*** on the instance variables of the class. static methods are declared as 'static'. To call the static methods, we need not create the object. we call a static method, as:

```
ClassName. methodname ();
```

Example-1: Write a program to perform addition of two numbers using static method without return statement.

```
class Addition
{
	static void add(double a,double b)
	{
		double c=a+b;
		System.out.println("Addition is "+c);
	}
}
class MethodDemo
{
	public static void main(String args[])
	{
		Addition.add(27,35);
	}
}
```

Output:

```
Addition is 62
```

Example-2: Write a program to perform addition of two numbers using static method with return statement.

```
class Addition
{
	static double add(double a,double b)
	{
		double c=a+b;
		return c;
	}
}
class MethodDemo
{
	public static void main(String args[])
	{
		double c= Addition.add(27,35);
	    System.out.println("Addition is "+c);

	}
}
```

Output:

```
Addition is 62
```

Q: why instance variables are not available to static methods?

Ans: *After executing static methods, JVM creates the objects. So, the instance variables are not available to static methods.*

Example: Write a JAVA program to implement method overloading.

Static Block:

- A static block is a block of statements declared as 'static'.

```
static {
      statements;
}
```

- JVM executes a static block on highest priority basis. This means JVM first goes to static block even before it looks for the `main()` method in the program.

Example-1: Write a program to test which one is-executed first by JVM, the static block or the static method.

```
class Demo
{
   static
         {
             System.out.println("Static block");
         }
         public static void main(String args[])
         {
             System.out.println("Static Method");
         }
}
```

Output:

```
Static block
Static Method
```

Example-2: Write a java program without `main()` method.

```
class Demo
{
      static
      {
            System.out.println("Static block");
      }
}
```

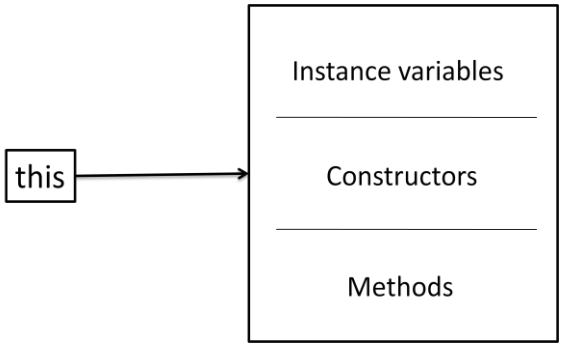
Output:

```
Static block
```

Q: Is it possible to compile and run a *Java program without writing* `main()` *method?*
Ans: Yes, it is possible by using a *static block in theJava program.*

'this' keyword:

- 'this' is a keyword that refers to the object of the class where it is used. In other words, 'this' refers the object of the present class.
- Generally, we write instance variables, constructors and methods in a class. All these members are referenced by 'this'. When an object is created to a class, a default reference is also created internally to the object.



Example: Write a program for 'this' keyword.

```
class Demo
{
        int x;
        Demo(int x)
        {
                this.x=x+3;
                System.out.println("x="+x);
                System.out.println("this.x="+this.x);
        }
        public static void main(String args[])
        {
                Demo d=new Demo(5);
        }
}
```

Output:

```
x=5
this.x=8
```

Recursion:

A method calling itself is known as recursive method', and that process is called 'recusrion'. It is possible to write recursive methods in Java. Let us, take an example to find the factorial value of the given number. Factorial value of number num is defined as: num * (num-1) * (num-2) * .....

Example: Write a java Program to find factorial of a given number using recursion.

```
import java.util.Scanner;
class Factorial
{
        static long fact(long num)
        {
                if(num==1 || num==0)
                        return 1;
                else
                {
                        return num*fact(num-1);
                }
        }
        public static void main(String args[])
        {
                Scanner sc=new Scanner(System.in);
```

```
                System.out.print("Enter number: ");
                long num=sc.nextInt();
                long f=Factorial.fact(num);
                System.out.println("Factorial is "+f);
        }
}
```

Nested Classes:

- Nested classes are the classes that contain other classes like inner classes. Inner class is basically a safety mechanism since it is hidden from other classes in its outer class.
- To make instance variables not available outside the class, we use 'private' access specifier before the variables. This is how we provide the security mechanism to variables. Similarly, in some cases we want to provide security for the entire class.
- In this case, can we use 'private' access specifier before the class? The problem is, if we use private access specifier before a class, the class is not available to the Java compiler or JVM. SO it is illegal to use 'private' before a class name in Java.
- But, private specifier is allowed before an inner class and thus it is useful to provide security for the entire inner class.
- An inner class is a class that is defined inside another class.
- Inner class is a safety mechanism.
- Inner class is hidden from other classes in its outer class.
- An object to inner class cannot be created in other classes.
- An object to inner class can be created only in its outer class.
- Inner class _can access the members of outer class directly.
- Inner class object and outer class objects are created in separate memory locations.

Example: Write a JAVA program to create the outer class BankAccount and the inner class Interest in it.

```
import java.util.Scanner;
class BankAccount
{
      double bal;
      BankAccount(double b)
      {
            bal=b;
      }
      void contact(double r)
      {
            Scanner sc=new Scanner(System.in);
            System.out.print("Enter Password: ");
            String password=sc.next();
            if(password.equals("cse556"))
            {
                  Interest in=new Interest(r);
                  in.calculateInterest();
            }
      }
```

```
        private class Interest
        {
              double rate;
              Interest(double r)
              {
                    rate=r;
              }
              void calculateInterest()
              {
                    double inter=bal*rate/100;
                    bal=bal+inter;
                    System.out.println("Updated Balance= "+bal);
              }
        }
        public static void main(String[] args)
        {
              BankAccount ba=new BankAccount(10000);
              ba.contact(9.5);
        }
  }
```

Output:

```
Enter Password: cse556
Updated Balance= 10950.0
```

Call by value:

- Call by Value means calling a method with a parameter as value. Through this, the argument value is passed to the parameter.
- While Call by Reference means calling a method with a parameter as a reference. Through this, the argument reference is passed to the parameter.
- In call by value, the modification done to the parameter passed does not reflect in the caller's scope while in the call by reference, the modification done to the parameter passed are persistent and changes are reflected in the caller's scope.

Example:

```
import java.util.*;
import java.lang.*;
class CallByValue
{
      int x;
      int y;
      void swap(int a,int b)
      {
            x = b;
            y = a;
      }
      public static void main(String[] args)
      {
            CallByValue cv = new CallByValue();
```

```
                cv.x = 3;
                cv.y = 5;
                System.out.println("Before Swapping "+cv.x+" "+cv.y);
                cv.swap(cv.x,cv.y);
                System.out.println("After Swapping "+cv.x+" "+cv.y);
        }
}
```

Output:

```
Before Swapping 3 5
After Swapping 5 5
```

Call by Reference:

Java uses only call by value while passing reference variables as well. It creates a copy of references and passes them as valuable to the methods. As reference points to same address of object, creating a copy of reference is of no harm. But if new object is assigned to reference it will not be reflected.

Example:

```
import java.util.*;
import java.lang.*;
class CallByReference
{
        int x;
        int y;
        void swap(CallByReference cr)
        {
                int temp = cr.x;
                cr.x = cr.y;
                cr.y = temp;
        }
        public static void main(String[] args)
        {
                CallByReference cr = new CallByReference();
                cr.x = 3;
                cr.y = 5;
                System.out.println("Before Swapping "+cr.x+" "+cr.y);
                cr.swap(cr);
                System.out.println("After Swapping "+cr.x+" "+cr.y);
        }
}
```

Output:

```
Before Swapping 3 5
After Swapping 5 5
```

Passing Object:

- While creating a variable of a class type, we only create a reference to an object. Thus, when we pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument.
- This effectively means that objects act as if they are passed to methods by use of call-by-reference.
- Changes to the object inside the method do reflect the object used as an argument.

Example:

```
class Student
{
	String Id;
	String name;
}
class Marks
{
	double cgpa;
	Marks(Student s,double cgpa)
	{
		this.cgpa=cgpa;
	}
	void display(Student s)
	{
		System.out.println(s.Id+" "+s.name+" "+this.cgpa);
	}
	public static void main(String[] args)
	{
		Student s=new Student();
		s.Id="556";
		s.name="Mothi";

		Marks m=new Marks(s,7.35);
		m.display(s);
	}
}
```

Output:

```
556 Mothi 7.35
```

INHERITANCE:

Deriving new classes from existing classes such that the new classes acquire all the features of existing classes is called inheritance. The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class). **extends** is the keyword used to inherit the properties from one class to another class.

Syntax:

```
class SuperClass
{
```

```
            ..................
            ..................
        }
        class SubClass extends SuperClass
        {
            ..................
            ..................
        }
```

Example: Write a JAVA Program to derive classes from one class to another class.

```
class SuperClass
{
      void add(int a,int b)
      {
            int c=a+b;
            System.out.println("Addition is "+c);
      }
}
class SubClass extends SuperClass
{
      void sub(int a,int b)
      {
            int c=a-b;
            System.out.println("Subtraction is "+c);
      }
}
class Demo{
 public static void main( String args[] )
 {
       SuperClass s1=new SuperClass();
       s1.add(5,6);
       SubClass s2=new SubClass();
       s2.sub(15,10);
       s2.add(15,27);
 }
}
```

Output:

```
Addition is 11
Subtraction is 5
Addition is 42
```

When an object to SubClass is created, it contains a copy of SuperClass within it. This means there is a relation between the SuperClass and SubClass objects. This is the reason why SuperClass members are available to SubClass. Note that we do not create A SuperClass object, but still a copy of it is available to SubClass object.

Q: *What is the advantage of Inheritance?*

Ans: *In inheritance, a programmer reuses the super class code without rewriting it, in creation of sub classes. So, developing the classes becomes very easy. Hence, the programmer's productivity is increased.*

The 'super' Keyword:

If we create an object to super class, we can access only the super class members, but not the sub class members. But if we create sub class object, all the members of both super and sub classes are available to it. This is the reason; we always create an object to sub class in inheritance. Sometimes, the super class members and sub class members may have same names. In that case, by default only sub class members are accessible. This is shown in the following example program.

Example:

```
class One
{
	int i=10;
	void show()
	{
		System.out.println("Super Class Method i: "+i);
	}
}
class Two extends One
{
	int i=20;
	void show()
	{
		System.out.println("Sub Class Method i: "+i);
	}
}
class Demo{
 public static void main( String args[] )
 {
	Two t=new Two();
	t.show();
 }
}
```

Output:

```
	Sub Class Method i: 20
```

Please observe that sub class method `t.show();` calls and executes only sub class method. And hence the instance variable i value 20 is displayed. To access super class members and methods by using 'super' keyword.

- super can be used to refer super class variables, as:

```
super.variableName
```

- super can be used to refer super class methods, as:

```
super.methodName()
```

- super can be used to refer super class constructor.

  We need not to call default constructor of the super class, as it is by default available to sub class. To call parameterized constructor, we can write:

  super(values)

Example: Write a program to access the super class method and instance variable by using super keyword from sub class.

```
class One
{
      int i=10;
      void show()
      {
            System.out.println("Super Class Method i: "+i);
      }
}
class Two extends One
{
      int i=20;
      void show()
      {
            System.out.println("Sub Class Method i: "+i);
            super.show();
            System.out.println("Super Class Variable i: "+super.i);
      }
}
class Demo{
 public static void main( String args[] )
 {
       Two t=new Two();
       t.show();
 }
}
```

Output:

```
Sub Class Method i: 20
Super Class Method i: 10
Super Class Variable i: 10
```

Example: Write a program to access parameterized constructor of the super class can be called from sub class using super keyword.

```
class One
{
      int i;
      One(int i)
      {
            this.i=i;
            this.i=this.i+2;
      }
```

```
}
class Two extends One
{
      int i;
      Two(int i)
      {
            super(i);
            this.i=i;
            System.out.println("Sub Class Variable i: "+i);
            System.out.println("Super Class Variable i: "+super.i);
      }
}
class Demo{
 public static void main( String args[] )
 {
       Two t=new Two(5);
 }
}
```

Output:

Sub Class Variable i: 5
Super Class Variable i: 7

final keyword:

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many contexts. Final can be:

a) variable
b) method
c) class

- The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable.
- It can be initialized in the constructor only.
- The blank final variable can be static also which will be initialized in the static block only.

a) Java final variable:

Example:

```
class Bike9{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
 obj.run();
 }
}
```

Output:

```
Compile Time Error
```

b) Java `final` method:

Example:

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

Output:

```
Compile time Error.
```

c) java `final` class

```
final class Bike
{
}
class Honda1 extends Bike
{
  void run()
  {
      System.out.println("running safely with 100kmph");
  }
  public static void main(String args[]){
  Honda1 honda= new Honda1();
  honda.run();
  }
}
```

Output:

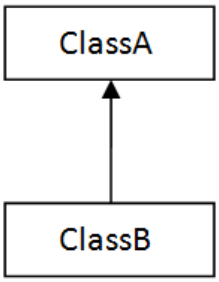
```
Compile time error
```

Types of Inheritance:

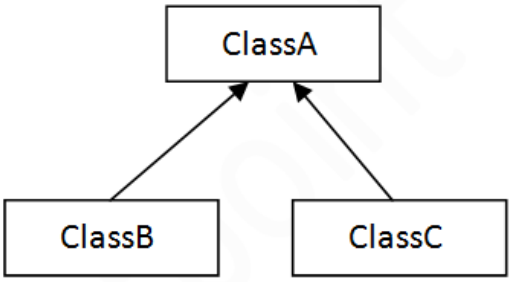
There are three types of inheritance in java.

1. Single Inheritance:

Producing sub classes from one super class is called single inheritance.



class B extends A



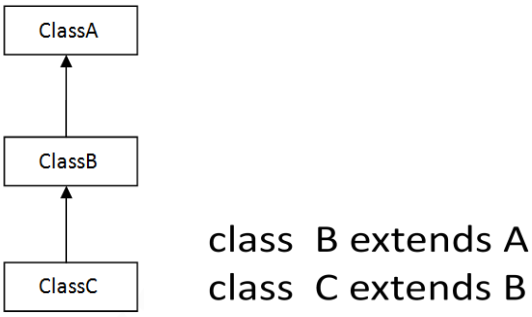
class B extends A
class C extends A

Example: Write a JAVA Program to derive classes from one class to another class.

```
class SuperClass
{
      void add(int a,int b)
      {
            int c=a+b;
            System.out.println("Addition is "+c);
      }
}
class SubClass extends SuperClass
{
      void sub(int a,int b)
      {
            int c=a-b;
            System.out.println("Subtraction is "+c);
      }
}
class Demo{
 public static void main( String args[] )
 {
       SuperClass s1=new SuperClass();
       s1.add(5,6);
       SubClass s2=new SubClass();
       s2.sub(15,10);
       s2.add(15,27);
 }
}
```

2. Multilevel Inheritance:

When a class extends a class, which extends anther class then this is called multilevel inheritance.



Example: Write a JAVA Program to illustrate multilevel inheritance.

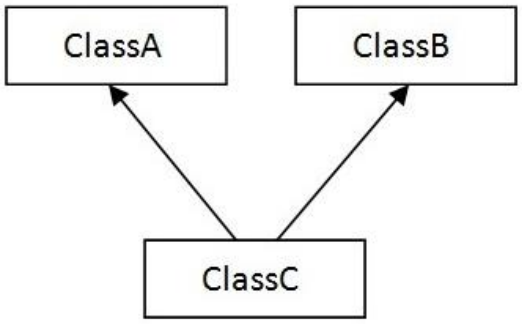
```
class A
{
	void show()
	{
		System.out.println("Method A");
	}
}
class B extends A
{
	void show()
	{
		super.show();
		System.out.println("Method B");
	}
}
class C extends B
{
	void show()
	{
		super.show();
		System.out.println("Method C");
	}
}
class MultiLevel
{
	public static void main(String[] args)
	{
		C c1=new C();
		c1.show();
	}
}
```

Output:

```
Method A
Method B
Method C
```

3. Multiple Inheritance:

Producing sub classes from multiple super classes is called multiple inheritance. In this case, there will be more than one super class and ·there can be one or more sub classes.



- Only single inheritance is available in java. There is no multiple inheritance in java. Multiple inheritance leads to confusion for the programmer. For example, class A has got a member x and class B has also got a member x. When another class C .extends both the classes, then there is a confusion regarding which copy of x is available in C.
- To overcome this problem JavaSoft people provide interface concept, expecting the programmers to achieve multiple inheritance by multiple interfaces.

Abstraction:

A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

- Abstract class
- Interface

Abstract class:

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 A{}
```

Abstract Method in Java

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

Example of abstract method

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

Example of Abstract class that has an abstract method

In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

Example:

```
abstract class Bike{
  abstract void run();
}
class Honda4 extends Bike{
      void run(){
            System.out.println("running safely");
      }
      public static void main(String args[]){
            Bike obj = new Honda4();
            obj.run();
      }
}
```

Understanding the real scenario of Abstract class

- Mostly, we don't know about the implementation class (which is hidden to the end user), and an object of the implementation class is provided by the factory method.
- A factory method is a method that returns the instance of the class. We will learn about the factory method later.
- In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

Example:

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

```
class Main{
      public static void main(String args[]){
            Circle1 s=new Circle1();
            s.draw();
      }
}
```

Object Classes:

- 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.
- The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, known as upcasting.
- Let's take an example, there is getObject() method that returns an object but it can be of any type like Employee,Student etc, we can use Object class reference to refer that object. For example:

  ```
  Object obj=getObject();
  ```

- The Object class provides some common behaviors to all the objects such as object can be compared, object can be cloned, object can be notified etc.

Method	**Description**
`public final Class getClass()`	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
`public int hashCode()`	returns the hashcode number for this object.
`public boolean equals(Object obj)`	compares the given object to this object.
`protected Object clone() throws CloneNotSupportedException`	creates and returns the exact copy (clone) of this object.
`public String toString()`	returns the string representation of this object.
`public final void notify()`	wakes up single thread, waiting on this object's monitor.
`public final void notifyAll()`	wakes up all the threads, waiting on this object's monitor.
`public final void wait(long timeout)throws InterruptedException`	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
`public final void wait(long timeout,int nanos)throws InterruptedException`	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
`public final void wait()throws InterruptedException`	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
`protected void finalize()throws Throwable`	is invoked by the garbage collector before object is being garbage collected.

Example:

```
class Test {
	public static void main(String[] args)
	{
		Object obj = new String("AID");
		Class c = obj.getClass();
		System.out.println(c.getName());
	}
}
```

Output:

```
Java.lang.String
```