Java Strings:

In java, string is basically an object that represents sequence of char values and string objects are immutable (cannot be modified).

Crating Strings: There are three ways to create strings in Java.

1. Create a string just by assigning a group of characters to a string type variable.

   String s; // declare a string type variable

   s=”PVPSIT”; //assign a group of characters to it

   Above two statements can be combined into

   String s= “PVPSIT”;

   In this case JVM creates an object to String class and stores the string “PVPSIT” in that object and is referenced with the name s.

2. We can create an object to String class by allocating memory using new operator. This is similar to object creating an object to any class.

   String s= new String(“PVPSIT”);

3. Another way of creating strings is by converting character arrays into

   strings. char arr[]={ ‘P’, ‘V’,’P’,’S’,’I’,’T’};

   Now, create a string object by passing the array name to it as

   String s=new String(arr);

   This means all the characters of array are copied into the string s. If we don’t want all characters of the array into the array, then we can mention which characters we need:

   String s=new String(arr, indice_location, size);

   String s=new String(arr,2,3);

String Class Methods:

Sl. No	Method	Description
1	char charAt(int index)	returns char value for the particular index
2	int length()	returns string length
5	String substring(int beginIndex)	returns substring for given begin index
6	String substring(int beginIndex, int endIndex-1)	returns substring for given begin index and end index
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 equalsIgnoreCase(String another)	Compares another string. It doesn't check case.
11	String[] split(String regex)	returns splitted string matching regex
12	int indexOf(int ch)	returns specified char value index
13	int indexOf(String substring)	returns specified substring 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	int compareTo(String str)	is used for comparing two strings lexicographically. Each character of both the strings is converted into a Unicode value for comparison. If both the strings are equal then this method returns 0 else it returns positive or negative value.
18	int compareToIgnoreCase (String str)	comparison of two strings irrespective of cases.

/* Java program to demonstration of String methods*/

```
class String_Demo
{
	public static void main(String arg[])
	{
		String s1="PVP Siddhartha";
		String s2=new String("Engineering");
		String s3=new String("College");

		System.out.println("First	:"+s1);
		System.out.println("Second	:"+s2);


		System.out.println("Length of first String			:"+s1.length());
		System.out.println("Concatenation of first and second	:"+s1.concat(s2));

		System.out.println("Concatenation of strings with + :"+s1+" "+s3);
		System.out.println("If string s1 starts with P or not	:"+s1.startsWith("P"));

		System.out.println("Extraction of substrings		:"+s1.substring(2));
		System.out.println("Extraction of substrings		:"+s1.substring(2,5));

		System.out.println("String into uppercase		:"+s1.toUpperCase());
```

```
                System.out.println("String into lower case          :"+s1.toLowerCase());

                String str = " Hello ";
                System.out.println("Use of Trim()                   :"+str.trim());
        }
}
```

/*Use of split() method*/

```
class String_Split
{
        public static void main(String ar[])
        {
                String str="PVP Siddhartha Inst. of Technology";

                //declare a string type array s to store pieces
                String s[];

                s=str.split(" "); //space

                for(int i=0;i<s.length;i++)
                        System.out.println(s[i]);
        }
}
```

Comparison of two strings: Relational operators <,<=,>,>=,==, != cannot be used to compare two strings. On the other hand, methods like compareTo(), compare() can be used.

```
                /* Comparison of two strings using == operator */

class CompareOperator
{
        public static void main(String arg[])
        {
                String s1="Hello";
                String s2=new String("Hello");

                if(s1==s2)
                        System.out.println("Same");
                else
                        System.out.println("Not Same");
        }
}
```

Output: Not Same

Explanation: When an object is created by JVM, it returns the memory address of the object as a hexadecimal number, which is called object reference. Whenever, a new object is

.

created, a new reference numbers is allotted to it. It means every object will have a unique reference.

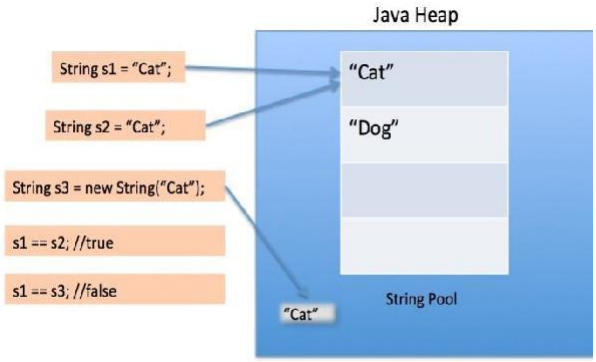
So, to compare two strings use **equals()** method.

```
class CompareEqual
{
	public static void main(String arg[])
	{
		String s1="Hello";
		String s2=new String("Hello");

		if(s1.equals(s2))
			System.out.println("true");
		else
			System.out.println("false");

	}
}
```

Output: true

Difference between == and equals() while comparing strings: == operator compares the reference of the string objects, not the content of the objects. Where as equals method compares the contents of the strings not object references.

String pool:




```
class String_Pool
{
	public static void main(String arg[])
	{
		String s1="Cat";
		String s2= "Cat";

		if(s1==s2)
			System.out.println("true");
		else
			System.out.println("false");

	}
}
```

Here, JVM creates a String object and stores “Welcome” in it. Observe that we are not using new operator to create the string. We are using = operator for this purpose. So after creating the String object, JVM uses a separate block of memory which is called **string constant pool** and stores the object**.**

In next statement when, s2 is executed by the JVM, it searches in the string constant pool to know whether the object with same content is already available or not. If so, copies the reference of that object to s2. So, we have same value in s1 and s2.

StringBuffer Class:

A String class objects are immutable and hence their contents cannot be modified. StringBuffer class objects are mutable, so they can be modified. Moreover the methods that directly manipulate the data of the object are not available in String class. Such methods are available in StringBuffer class.

Creating StringBuffer objects:

There are two ways to create StringBuffer object, and fill the object with string.

1. We can create a StringBuffer object by using new operator and pass the string to the object, as:

```
Ex 1.  StringBuffer sb=new StringBuffer("Hello");
Ex 2.  String str=new String("Hello");
       StringBuffer sb=new StringBuffer(str);
```

2. Second way is, allocate the memory to the StringBuffer object with new operator and later store string data into it.

```
StringBuffer sb=new StringBuffer();
```

Here, we are creating a StringBuffer object and empty object and not passing any string to it. In this case, object will be created with default capacity of 16 characters.

```
StringBuffer sb=new StringBuffer(50); //50 characters
sb.append("Hello");
        or
sb.insert(0,"Hello");
```

Methods:

Sl. No	Method	Description
1	append(String s)	is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc
2	insert(int offset, String s)	is used to insert the specified string with this string at the specified position.
3	replace(int startIndex, int endIndex, String str)	is used to replace the string from specified startIndex and endIndex.
4	delete(int startIndex, int endIndex)	is used to delete the string from specified startIndex and endIndex
5	reverse()	is used to reverse the string

6	ensureCapacity(int minimumCapacity)	is used to ensure the capacity at least equal to the given minimum.
7	charAt(int index)	is used to return the character at the specified position.
8	length()	is used to return the length of the string i.e. total number of characters
9	String substring(int beginIndex)	is used to return the substring from the specified beginIndex.
10	substring(int beginIndex, int endIndex)	is used to return the substring from the specified beginIndex and endIndex.

Example Program:

```
class StringBufferDemo
{
	public static void main(String arg[])
	{
		StringBuffer sb=new StringBuffer();
		System.out.println("Use of capacity():"+sb.capacity());

		StringBuffer sb1=new StringBuffer("Uni");
		System.out.println("Use of append():"+sb1.append("versity"));
		System.out.println("Use of insert():"+sb1.insert(0, "JNTU "));
		System.out.println("use of delet()"+sb1.delete(0,4));
		System.out.println("Use of length():"+sb1.length());

		System.out.println("Use of reverse():"+sb1.reverse());
		System.out.println("Use of capacity():"+sb1.capacity());
		System.out.println("Use of reverse():"+sb1.reverse());
		System.out.println("Use of indexOf():"+sb1.indexOf("i"));
		System.out.println("Use of lastIndexOf:"+sb1.lastIndexOf("i"));
		System.out.println("Use of substring(int):"+sb1.substring(4));
		System.out.println("Use of substring(int,int):"+sb1.substring(0,4));
		System.out.println(sb1);
		System.out.println("Use of replace():"+sb1.replace(0,3,"ANU "));
	}
}
```

Output:

```
Use of capacity():16
Use of append():University
Use of insert():JNTU University
use of delet() University
Use of length():11
Use of reverse():ytisrevinU
Use of capacity():19
Use of reverse(): University
Use of indexOf():3
Use of lastIndexOf:8
Use of substring(int):versity
Use of substring(int,int): Uni
 University
Use of replace():ANU iversity
```

StringTokenizer Class:

The **java.util.StringTokenizer** class allows you to break a string into pieces called tokens. These tokens are then stored in the StringTokenizer object.

The code to create an object to StringTokenizer class is

1. StringTokenizer obj=new StringTokenizer(String str);
   Splits the given string into tokens based on default delimiter space.
2. StringTokenizer obj=new StringTokenizer(String str, String delimter); Splits the given string into tokens based on the delimeter.
   Ex. StringTokenizer obj=new StringTokenizer("pvp,Siddhartha,inst,tech","");
   Splits the above string into tokens based on the delimeter comma.

Methods:

S. No	Method	Description
1	boolean hasMoreTokens()	checks if there is more tokens available.
2	String nextToken()	returns the next token from the StringTokenizer object.
3	boolean hasMoreElements()	same as hasMoreTokens() method.
4	Object nextElement()	same as nextToken() but its return type is Object.
5	int countTokens()	returns the total number of tokens.

Example:

```
import java.util.StringTokenizer;
class Simple
{
	public static void main(String args[])
	{
		StringTokenizer st = new StringTokenizer("my,name,is,pvpsit",",");
		while (st.hasMoreTokens())
		{
			System.out.println(st.nextToken());
		}
	}
}
```

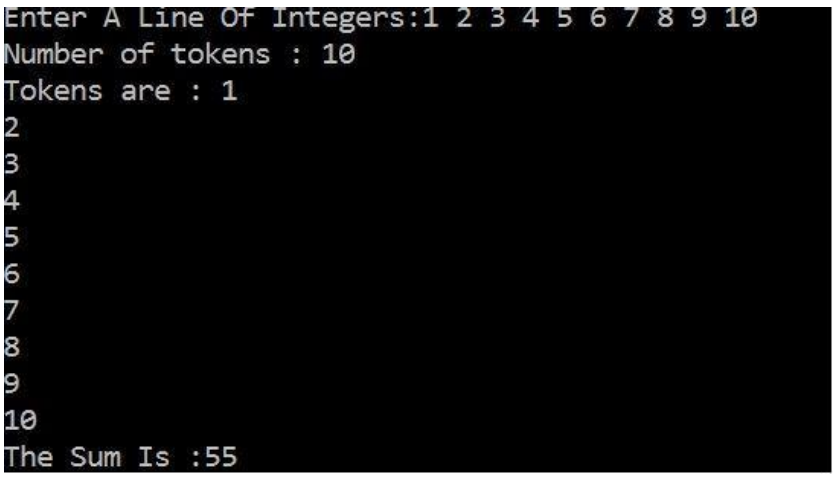
Output:

```
my
nameDifference between spilt and StringTokenizer:
is
pvpsit
```

/* Write a Java Program that reads a line of integers, and then displays each integer, and the sum of all the integers (Use StringTokenizer class of java.util) */

```
import java.util.*;

class StringTokenizerEx
{
   public static void main(String[] args)
   {
        Scanner s = new Scanner(System.in);
      System.out.print("\nEnter A Line Of Integers:");

      String line = s.nextLine();

      StringTokenizer st = new StringTokenizer(line);
      System.out.println("Number of tokens : "+st.countTokens());
      int sum = 0;
      System.out.print("Tokens are : " );
      while (st.hasMoreTokens())
      {
         int i = Integer.parseInt(st.nextToken());
         System.out.println(i);
         sum = sum + i;
      }
      System.out.println("The Sum Is :" +sum);
    }
}
```

Output:



Difference between StringTokenizer and split():

In StringTokenizer, the delimiter is just one character long. You supply a list of characters that count as delimiters, but in that list, each character is a single delimiter. With split(), the delimiter is a regular expression, which is something much more powerful (and more complicated to understand). It can be any length.

```
import java.util.StringTokenizer;
class D_Split_ST
{
	public static void main(String args[])
	{
		StringTokenizer st = new StringTokenizer("This is another example",
		"char"); //delimiters are individual characters such as c, h, a r

		while (st.hasMoreTokens())
		{
			System.out.println(st.nextToken());

		}

		String str="This is another example";

		//declare a string type array s to store pieces
		String s[];

		s=str.split("char"); // here the delimiter is char

		for(int i=0;i<s.length;i++)
			System.out.println(s[i]);

	}
}
```

Output:

```
T
is is
not
e
 ex
mple
This is another example
```

/* Write a Java program that checks whether a given string is a palindrome or not. Ex MALAYALAM is a palindrome. */

```
import java.io.*;
class Palindrome
{
	public static void main(String args[])
	{
		String str1=new String(args[0]);
		StringBuffer str2=new StringBuffer(str1);
		str2.reverse();

		for(int i=0;i<str2.length();i++)
		{
			if(str1.charAt(i)!=str2.charAt(i))
			{
				System.out.println(str1+" is not palindrome");
				System.exit(0);
			}
		}
		System.out.println(str1+" is palindrome");
	}
}
```

/* Write a Java program for sorting a given list of names in ascending order.*/

```
import java.util.*;
class SortEx
{
	public static void main(String arg[ ])
	{
		Scanner s=new Scanner(System.in);

		System.out.println("Enter the size");
		int n=s.nextInt();

		String array[]=new String[n];

		System.out.println("Enter names");
		for(int i=0;i<n;i++)
		{
			array[i]= s.next();
		}

		for(int i=0;i<array.length-1;i++)
		{
			for(int j=i+1;j<array.length;j++)
			{
				if(array[i].compareToIgnoreCase(array[j])>0)
				{
					String Temp = array[i] ;
					array[i] = array[j] ;
```

```
                                        array[j] = Temp ;
                                }
                        }
                }
                System.out.println("After Sorting");
                for(int i=0;i<n;i++)
                {
                        System.out.println(array[i]);
                }
        }
}
```

Inheritance:

- Inheritance can be defined as the process where one class acquires the properties of another class
- 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).
- Inheritance represents the IS-A relationship, also known as parent-child relationship.
- Advantages of inheritance:
    - Code reusability
    - Used in method overriding (so runtime polymorphism can be achieved).

Syntax:

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

The extends keyword indicates that you are making a new class that derives from an existing class.

Example:

```
// A simple example of inheritance.
// Create a superclass.
class A
{
        int i, j;
        void showij()
        {
                System.out.println("i and j: " + i + " " + j);
        }
}
// Create a subclass by extending class A.
class B extends A
{
        int k;
        void showk()
        {
                System.out.println("k: "+ k);
        }
void sum()
{
```

```
        System.out.println("i+j+k: " + (i+j+k));
        }
}
class SimpleInheritance
{
        public static void main(String args [])
        {
                B subOb = new B();

                /* The subclass has access to all public members of its superclass. */
                subOb.i  =  7;
                subOb.j  =  8;
                subOb.k = 9;

                System.out.println("Contents of subOb: ");
                subOb.showij();
                subOb.showk();
                System.out.println();
                System.out.println("Sum of i, j and k in subOb:");
                subOb.sum();
        }
}
```

```
D:\Materials\JAVA Material\Unit 2\Inheritance>java SimpleInheritance
Contents of subOb:
i and j: 7 8
k: 9

Sum of i, j and k in subOb:
i+j+k: 24
```

Member Access and Inheritance

Although a subclass includes all of the members of its super class, it cannot access those members of the super class that have been declared as private. For example, consider the following simple class hierarchy:

```
/* In a class hierarchy, private members remain private to their class. This program contains an error and will not compile. */
        class A
        {
                int i; // public by default
                private int j;  //Private to A
```

```
        void setij(int x, int y)
        {
                i = x;
                j = y;
        }
}
class B extends A
{
        int total;
        void sum()
        {
                total = i + j;  //A's j is not accessible here
        }
}
class SimpleInheritance2
{
        public static void main(String args[])
        {
                B subOb = new B();
                subOb.setij(10,  12);
                subOb.sum();
                System.out.println("Total is " + subOb.total);
        }
}
```

```
D:\Materials\JAVA Material\Unit 2\Inheritance>javac SimpleInheritance2.java
SimpleInheritance2.java:22: error: j has private access in A
                total = i + j;  //A's j is not accessible here
                            ^
1 error
```

A super class variable can reference a subclass object:

A reference variable of a super class can be assigned a reference to any subclass derived from that super class.

Ex:

```
        class A
        {
                void callMe()
                {
                        System.out.print("Hello");
                }
        }
```

```
class B extends A
{
        void callMe()
        {
                System.out.print("Hi ");
        }
 }
class Reference
{
        public static void main(String args[])
        {
                A ref;
                B b = new B();
                ref = b;
                ref.callMe();
        }
}
```

Output: Hi

Using super

Whenever the derived class is inherits the base class features, there is a possibility that base class features are similar to derived class features and JVM gets an ambiguity. To overcome this, super is used to refer super class properties.

The super keyword in java is a reference variable that is used to refer parent class. The keyword “super” came into the picture with the concept of Inheritance. It is majorly used in the following contexts:

- super is used to refer immediate parent class instance variable.
  ```
  super.parent_instance_variable_name;
  ```
- super is used to invoke immediate parent class method
  ```
  super.parent_class_method_name();
  ```
- super is used to invoke immediate parent class constructor.
  ```
  super(arglist); // parameterized constructor
  super(); //default
  ```

Usage 1. - Using super to refer super class property

```
class Vehicle
{
        int speed=50;
}
class Bike extends Vehicle
{
         int speed=100;
         void display()
         {
             System.out.println("Vehicle Speed:"+super.speed);//will print speed of vehicle
             System.out.println("Bike Speed:"+speed);//will print speed of bike
         }
}
class SuperVarible
{
            public static void main(String args[])
           {
                    Bike b=new Bike();
                    b.display();
            }
}
```

Output:

```
D:\Materials\JAVA Material\Unit 2\UseofSuper>java SuperVarible
Vehicle Speed:50
Bike Speed:100
```

Usage 2. - super is used to invoke immediate parent class method

```
class Student
{
         void message()
         {
                  System.out.println("Good Morning Sir");
         }
}

class Faculty extends Student
{
```

```
        void message()
        {
                System.out.println("Good Morning Students");
        }
        void display()
        {
                message();//will invoke or call current class message() method
                super.message();//will invoke or call parent class message() method
        }
}
class SuperClassMethod
{
        public static void main(String args[])
        {
                Faculty f=new Faculty();
                f.display();
        }
}
```

Output:

```
D:\Materials\JAVA Material\Unit 2\UseofSuper>java SuperClassMethod
Good Morning Students
Good Morning Sir
```

Usage 3. - super is used to invoke immediate parent class constructor

```
class Vehicle
{
          Vehicle( )
          {
                System.out.println("Vehicle is created");
          }
}

class Bike extends Vehicle
{
           Bike()
           {
                super( );//will invoke parent class constructor
                System.out.println("Bike is created");
           }

}
class SuperClassConst
{
```

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

Output:

```
D:\Materials\JAVA Material\Unit 2\UseofSuper>java SuperClassConst
Vehicle is created
Bike is created
```

Calling Constructors:

Constructors are called in order of derivation, from superclass to subclass. Further, since super( ) must be the first statement executed in a subclass' constructor, this order is the same whether or not super( ) is used. If super( ) is not used, then the default or parameterless constructor of each superclass will be executed. The following program illustrates when constructors are executed:

```
// Create a super class.
class A {
    A() {
        System.out.println("Inside A's constructor.");
    }
}
// Create a subclass by extending class A.
class B extends A
{
    B() {
        System.out.println("Inside B's constructor.");
    }
}
// Create another subclass by extending B.
class C extends B {
    C() {
        System.out.println("Inside C's constructor.");
    }
}
class CallingConstructor
{
    public static void main(String args[])
    {
```

```
            C c = new C();
        }
}
```

Output:

```
D:\Materials\JAVA Material\Unit 2\UseofSuper>java CallingConstructor
Inside A's constructor.
Inside B's constructor.
Inside C's constructor.
```

Method Overriding:

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass.

When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

Example:

```
// Method overriding.
class A
{
        int i, j;
        A(int a, int b)
        {
                i = a;
                j = b;
        }
        // display i and j
        void show()
        {
                System.out.println("i and j: " + i + " " + j);
        }
}

class B extends A
{
        int k;
        B(int a, int b, int c)
        {
                super(a, b);
```

```
            k = c;
        }
        // display k – this overrides show() in A
        void show()
        {
            System.out.println("k: " + k);
        }
}
class Override
{
        public static void main(String args[]) { B
            subOb = new B(1, 2, 3);
            subOb.show(); // this calls show() in B
        }
}
```

Output:

```
D:\Materials\JAVA Material\Unit 2\UseofSuper>java Override
k: 3
```

Dynamic Method Dispatch:

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

Example: //A Superclass Variable Can Reference to a Subclass Object

```
class A
{
        void callMe()
        {
            System.out.println("Inside A");
        }
}
class B extends A
{
        void callMe()
        {
            System.out.println("Inside B");
        }
}

class C extends B
{
```

```
        void callMe()
        {
                System.out.println("Inside C");
        }
}
class DynamicMethodDispatch
{
        public static void main(String args[])
        {
                A a=new A();
                B b = new B();
                C c = new C();
                A ref;
                ref = a;
                ref.callMe();

                ref = b;
                ref.callMe();

                ref=c;
                ref.callMe();
        }
}
```

Output:

```
D:\Materials\JAVA Material\Unit 2\Inheritance>java DynamicMethodDispatch
Inside A
Inside B
Inside C
```

Applying Method Overriding:

```
import java.util.*;

class Student
{
        int n;
        String name;
        void read()
        {
                Scanner s=new Scanner(System.in);
                System.out.println("Enter no and Name");
                n=s.nextInt();
                name=s.nextLine();

        }
```

```
	void show()
	{
		System.out.println("No:"+n);
		System.out.println("Name:"+name);
	}
}
class ITStudent extends Student
{
	void read()
	{
		super.read();
	}
	void show()
	{
		super.show();
	}
}
class CSEStudent extends Student
{
	void read()
	{
		super.read();
	}
	void show()
	{
		super.show();
	}

}
class ECEStudent extends Student
{
	void read()
	{
		super.read();
	}
	void show()
	{
		super.show();
	}
}
class MainMethod
{
	public static void main(String ar[])
	{
		ITStudent it=new ITStudent();
		CSEStudent cse=new CSEStudent();
```

```
        ECEStudent ece=new ECEStudent();

        Student sref;

        sref=it;
        sref.read();
        sref.show();

        sref=cse;
        sref.read();
        sref.show();

        sref=ece;
        sref.read();
        sref.show();
    }
}
```

Abstract Class

- An abstract class is a class that contains one or more abstract methods.
- An abstract method is a method without method body.

  Syntax:

  abstract return_type method_name(parameter_list);
- An abstract class can contain instance variables, constructors, concrete methods in addition to abstract methods.
- All the abstract methods of abstract class should be implemented in its sub classes.
- If any abstract method is not implemented in its subclasses, then that sub class must be declared as abstract.
- We cannot create an object to abstract class, but we can create reference of abstract class.
- Also, you cannot declare abstract constructors or abstract static methods.

Example 1: Java program to illustrate abstract class.

```
abstract class MyClass
{
        abstract void calculate(double x);
}
class Sub1 extends MyClass
{
        void calculate(double x)
```

```
        {
                System.out.println("Square  :"+(x*x));
        }
}
class Sub2 extends MyClass
{
        void calculate(double x)
        {
                System.out.println("Square Root :"+Math.sqrt(x));
        }
}
class Sub3 extends MyClass
{
        void calculate(double x)
        {
                System.out.println("Cube :"+(x*x*x));
        }
}
class AC
{
        public static void main(String arg[])
        {
                Sub1  obj1=new  Sub1();
                Sub2  obj2=new  Sub2();
                Sub3 obj3=new Sub3();

                obj1.calculate(20);
                obj2.calculate(20);
                obj3.calculate(20);
        }
}
```

Example 2: Java program to illustrate abstract class.

```
// A Simple demonstration of abstract.
abstract class A
{
        abstract void callme();
        // concrete methods are still allowed in abstract classes
        void callmetoo() {
                System.out.println("This is a concrete method.");
        }
}
 class B extends A
{
        void callme()
```

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

class AbstractDemo
{
        public static void main(String args[])
        {
                B b = new B();
                b.callme();
                b.callmetoo();
        }
}
```

Uses of Final:

Final can be used in three ways:

- To prevent modifications to the instance variable
- To Prevent method overriding
- To prevent inheritance

Use 2: To prevent method overriding

```
class Vehicle
{
        final void run()
        {
                System.out.println("running");
        }
}
class Bike extends Vehicle
{
        void run()
        {
                System.out.println("running safely with 100kmph");
        }
}

class FinalMethod
{
        public static void main(String args[])
        {
                Bike b= new Bike();
                b.run();
```

```
    }
}
```

Error Information:

```
D:\Materials\JAVA Material\Unit 2\Inheritance>javac FinalMethod.java
FinalMethod.java:10: error: run() in Bike cannot override run() in Vehicle
        void run()
             ^
  overridden method is final
1 error
```

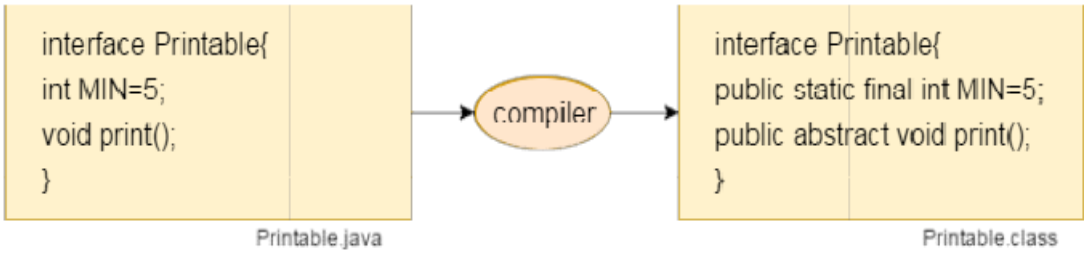
Use 3: To prevent inheritance

```
final class Vehicle
{
    void run()
    {
        System.out.println("running");
    }
}
class Bike extends Vehicle
{
    void run()
    {
        System.out.println("running safely with 100kmph");
    }
}

class FinalClass
{
    public static void main(String args[])
    {
        Bike b= new Bike();
        b.run();
    }
}
```

```
D:\Materials\JAVA Material\Unit 2\Inheritance>javac FinalClass.java
FinalClass.java:8: error: cannot inherit from final Vehicle
class Bike extends Vehicle
                   ^
1 error
```

Interfaces:

- A named collection of method declarations.
- A Java interface is a collection of constants and abstract methods
- Since all methods in an interface are abstract, the abstract modifier is usually left off
- Using interface, you can specify what a class must do, but not how it does.
- Interface fields are public, static and final by default, and methods are public and abstract.



Advantages of interfaces:

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritances.

Syntax:

```
access_specifier interface interfae_name
{
        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;
}
```

Implementing Interfaces:

- Once an interface has been defined, one or more classes can implement that interface.
- To implement an interface, include the implements clause in a class definition, and then create the methods defined by the interface.
- The general form of a class that includes the implements clause looks like this:

```
class classname [extends superclass] [implements interface1 [,interface2...]] {
        // class-body
}
```

- If a class implements more than one interface, the interfaces are separated with a comma.
- The methods that implement an interface must be public. Also, the type signature of implementing method must match exactly the type signature specified in interface definition.

Example 1: Write a java program to implement interface.

```
interface Moveable
{
        int AVG_SPEED=30;
        void Move();
}
class Move implements Moveable
{
         void Move(){
                System .out. println ("Average speed is:  "+AVG_SPEED  );
         }
}
class Vehicle
{
         public static void main (String[] arg)
         {
                Move m = new Move();
                m.Move();
         }
}
```

Example 2: Write a java program to implement interface.

```
 interface Teacher
{
        void display1();
}
interface Student
{
        void display2();
}
class College implements Teacher, Student
{
        public void display1()
        {
                System.out.println("Hi I am Teacher");
        }
        public void display2()
        {
                System.out.println("Hi I am Student");
        }
}
class CollegeData
{
        public static void main(String arh[])
        {
                College c=new College();
                c.display1();
                c.display2();
        }
}
```

Accessing implementations through interface references:

We can declare variables as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be referred to by such a variable. When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces. The method to be executed is looked up dynamically at run time,

allowing classes to be created later than the code which calls methods on them.

Ex:

```
interface Test {

   void call();
}

class InterfaceTest implements Test {

   public void call()
   {
      System.out.println("call method called");
   }
}

public class IntefaceReferences {

        public static void main(String[] args)
        {
                Test f ;
                InterfaceTest it= new InterfaceTest();
                f=it;
                f.call();
            }
}
```

Variables in Interfaces:

We can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values. When we include that interface in a class (that is, when you "implement" the interface), all of those variable names will be in scope as constants. (This is similar to using a header file in C/C++ to create a large number of #defined constants or const declarations.) If an interface contains no methods, then any class that includes such an interface doesn't actually implement anything. It is as if that class were importing the constant fields into the class name space as final variables.

Example: Java program to demonstrate variables in interface.

```
interface left
{
        int i=10;
}
interface right
```

```
{
        int i=100;
}
class Test implements left,right
{
        public static void main(String args[])
        {
                System.out.println(left.i);//10 will be printed
                System.out.println(right.i);//100 will be printed*/
        }
}
```

Interfaces can be extended:

One interface can inherit another by use of the keyword extends. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

Example: Java program to demonstrate interfaces can be extended with extend keyword.

```
interface Teacher
{
        void display1();
}
interface Student
{
        void display2();
}
interface T_S extends Teacher, Student
{
        void display3();
}
 class College implements T_S
{
        public void display1()
        {
                System.out.println("Hi I am Teacher");
        }
        public void display2()
        {
                System.out.println("Hi I am Student");
        }
        public void display3()
        {
                System.out.println("Hi I am Teacher_Student");
```

```
        }
}
class Class_Interface
{
        public static void main(String arh[])
        {
                College c=new College();
                c.display1();
                c.display2();
                c.display3();
         }

}
```

Example 2: Java program to implement interface and inheriting the properties from a class.

```
interface Teacher
{
        void display1();
}
class Student
{
        void display2()
        {
                System.out.println("Hi I am Student");
        }
}
class College extends Student implements Teacher
{
        public void display1()
        {
                System.out.println("Hi I am Teacher");
        }
}
class Interface_Class
{
        public static void main(String arh[])
        {
                College c=new College();
                c.display1();
                c.display2();
        }
}
```

Difference between Interface and Abstract class:

Abstract Class	Interface
Contains some abstract methods and some concrete methods	Only abstract methods
Contains instance variables	Only static and final variables
Doesn't support multiple inheritance	Supports
public class Apple extends Food { … }	public class Person implements Student,Athlete,Chef { … }

Use of interfaces :

To reveal an object's programming interface (functionality of the object) without revealing its implementation.

- This is the concept of encapsulation.
- The implementation can change without affecting the caller of the interface.
- To have unrelated classes implement similar methods (behaviors).
- One class is not a sub-class of another.
- To model multiple inheritance.
- A class can implement multiple interfaces while it can extend only one class.

Java Nested Interfaces

✓ An interface i.e. declared within another interface or class is known as nested interface.

✓ The nested interfaces are used to group related interfaces so that they can be easy to maintain

✓ The nested interface must be referred by the outer interface or class. It can't be accessed directly.

Points to remember for nested interfaces :

i) Nested interface must be public if it is declared inside the interface but it can have any access modifier if declared within the class.

ii) Nested interfaces are declared static implicitly.

Syntax of nested interface which is declared within the interface :

```
interface interface_name{
 ...
 interface nested_interface_name{
  ...
 }
}
```

Syntax of nested interface which is declared within the class :

```
class class_name{
 ...

 interface nested_interface_name{
  ...
 }
}
```

Example using Nested Interfaces (Inside an interface):

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

```
}
class TestNestedInterface1  implements  Showable.Message
{
 public void msg()
{
 System.out.println("Hello nested interface");
}

 public static void main(String args[])
{
  Showable.Message message=new TestNestedInterface1();//upcasting here
  message.msg();
 }
}
```

Example for nested interface (inside a class) :

```
class A
 {
  interface Message
  {
   void msg();
  }
}

Class NestedIntTest2 implements A.Message
{
 public void msg()
{
System.out.println("Hello nested interface");
}
   public static void main(String args[])
 {
  A.Message message=new NestedIntTest2();   //upcasting here
  message.msg();
 }
}
```