String

PresentationPoint

Java strings are class objects and implemented using two classes namely String and StringBuffer.

Creating a string :

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

char ch[] = {'a','b','c'};
String s = new String(ch);

char ch[] = {'a','b','c','d','e','f'};
String s = new String(ch,2,3 );
```

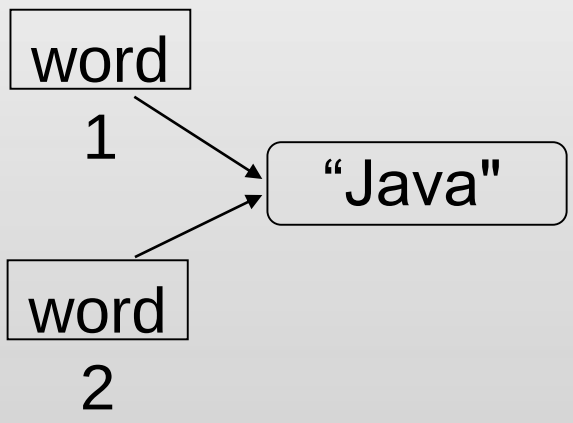
OUTPUT: cde

Immutability

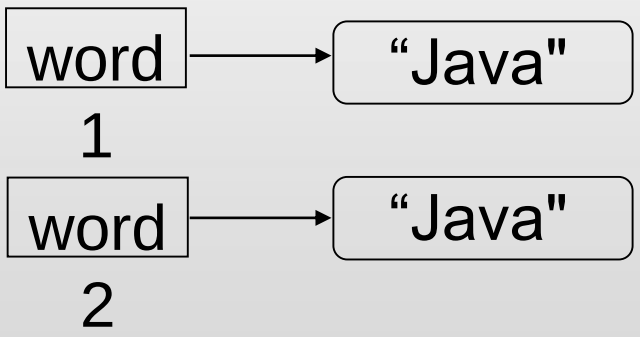
- Once created, a string cannot be changed: none of its methods changes the string.
- Such objects are called *immutable*.
- Immutable objects are convenient because several references can point to the same object safely: there is no danger of changing an object through one reference without the others being aware of the change.

PresentationPoint

```
String word1 = "Java";
String word2 = word1;
```




```
String word1 = “Java";
String word2 = new String(word1);
```



PresentationPoint

Special string syntax:

String s=“mit”;

String Concatenation

+ operator is used for concatenation.

String s = “Hello”+”world”;

Ex :

```
String s1 =“four : “+2+2;
String s2= “four : “+(2+2);


System.out.println(s1);
System.out.println(s2);
```

String Methods

int length( )

Ex:

```
String   s = “computer”;
int a = s.length();
```

String Methods

String toLowerCase( )

String toUpperCase( )

```
class prg2
{
    public static void main(String args[])
     {
            String s = "abcabcabc";
            System.out.println(s.toUpperCase());
            System.out.println(s.toLowerCase());
            System.out.println(s.length());
      }
}
```

String Methods

char charAt(int *where)*

```
class prg
{
   public static void main(String args[])
   {
            String s1 = "Mangalore";
            System.out.println(s1.charAt(3));
   }
}
```

String Methods

String replace(char *original, char replacement)*

```
class prg
{
   public static void main(String args[])
   {
            String s1 = "Mangalore";
            System.out.println(s1.replace('M','B'));
    }
}
```

String Methods

boolean equals(String *str)*

boolean equalsIgnoreCase(String *str)*

```
class prg
{
    public static void main(String args[])
    {
        String s1 = “Bangalore";
        String s2 = “bangalore";
        if(s1.equals(s2))
                System.out.println("same place");
        else
                System.out.println("different place");

    }
}
```

```
class prg
{
    public static void main(String args[])
    {
        String s3 = “Mit";
        String s4 = “mit";
        if(s3.equalsIgnoreCase(s4))
                System.out.println("same college");
        else
                System.out.println("different college");
  }
}
```

String Methods

int compareTo(String *str)*

Here, *str is the* String being compared with the invoking String. The result of the comparison is returned and is interpreted as shown here:

Value	Meaning
Less than zero	The invoking string is less than *str.*
Greater than zero	The invoking string is greater than *str.*
Zero	The two strings are equal

```
class prg
{
   public static void main(String args[])
   {
            String s1="computer";
            String s2="electronics";
            System.out.println(s1.compareTo(s2));

            String s3 = “mit";
            String s4 = “mit";
            System.out.println(s3.compareTo(s4));
   }
}
```

String Methods

String substring(int *startIndex)*

String substring(int *startIndex, int endIndex)*

Here, *startIndex specifies the beginning index, and endIndex specifies the stopping* point.

The string returned contains all the characters from the beginning index, up to, but not including, the ending index.

```
class prg
{
   public static void main(String args[])
   {
            String s = "abcdefgh";
            System.out.println(s.substring(3));
            System.out.println(s.substring(3,5));
   }
}
```

OUTPUT :

defgh

de

Int length()

String toLowerCase()

String toUpperCase()

char charAt(int p)

String replace(char c1, char c2)

boolean equals(String s2)

boolean equalsIgnoreCase(String s2)

Int compareTo(String s)

String substring(int startIndex)

String substring(int startIndex, int endIndex)

Count number of ‘a’ present in a given string

Replace all occurrences of ‘a’ with ‘b’;

Replace all character which is same as first character with last character of a given string.

String Methods

String concat(String *str)*

String trim( )

```
class prg
{    public static void main(String args[])
     {
            String s1=“MIT";
            String s2="Manipal";
            System.out.println(s1.concat(s2));

            String s3 = s1.concat(s2);

            System.out.println(s1);
            System.out.println(s3);
  } }
```

String Methods

```
int    indexOf(char ch);

int    lastIndexOf(char  ch);


int     indexOf(String st);

int     lastIndexOf(String  st);
```

```
class prg
{       public static void main(String args[])
     {
              String s = “object oriented programming”
              System.out.println(s.indexOf(‘e'));
              System.out.println(s.lastIndexOf(‘e'));

              String s2=“Service and Excellence”;
            System.out.println(s2.indexOf(“ce"));
            System.out.println(s2.lastIndexOf(“ce"));
   }
}
```

OUTPUT :

3

13

5

20

String Methods

```
int     indexOf(int ch, int fromIndex);

int     lastIndexOf((int ch, int fromIndex);

int     indexOf(String st, int fromIndex);

int     lastIndexOf(String st, int fromIndex);
```

fromIndex specifies the index at which point the search begins.

For **indexOf( ),** the search runs from startIndex to the end of the string.

For **lastIndexOf( ),** the search runs from fromIndex to zero.

```
class prg
{
   public static void main(String args[])
   {
            String s = "abcabcabc";
            System.out.println(s.indexOf('a',3));
             System.out.println(s.lastIndexOf(‘c',2));
            System.out.println(s.lastIndexOf(‘c',7));

   }
}
```

OUTPUT

3

2

5

```
class prg
{
   public static void main(String args[])
   {
                        String s = "abcabcabc";
                        System.out.println(s.indexOf('c'));
                        System.out.println(s.indexOf('c',3));
                        System.out.println(s.lastIndexOf('c'));
                        System.out.println(s.lastIndexOf('c‘,7));
   }
}
```

```
2
5
8
5
```

String Methods

boolean startsWith(String *str)*

boolean endsWith(String *str)*

equals( ) Versus ==

- **equals( ) method compares the** characters inside a **String object.**

- **The == operator compares two object references to** see whether they refer to the same instance.

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

OUTPUT

true

false

Check whether a given character is present in a given string or not

Count number of vowels present in a given string

Int length()

String toLowerCae()

String toUpperCase()

char charAt(int p)

String replace(char c1, char c2)

boolean equals(String s2)

Boolean equalsIgnoreCase(String s2)

Int compareTo(String s)

String substring(int startIndex)

String substring(int startIndex, int endIndex)

String trim()

String concat(String s)

boolean startsWith(String s);

Boolean endswith(Strings s)

Boolean equals(Strig s)

int indexOf(int ch);

int lastIndexOf(int cha);

int indexOf(String st);

int lastIndexOf(String st);

int indexOf(int ch, int fromIndex);

int lastIndexOf((int ch, int fromIndex);

int indexOf(String st, int fromIndex);

int lastIndexOf(String st, int fromIndex);

getChars( )

void getChars(int *sourceStart*, int *sourceEnd*, char *target*[ ], int *targetStart*)

sourceStart specifies the index of the beginning of the substring,
sourceEnd specifies an index that is one past the end of the desired substring.

Thus, the substring contains the characters from *sourceStart* through *sourceEnd*–1.

The array that will receive the characters is specified by *target*. The index within *target* at which the substring will be copied is passed in *targetStart*.

getChars( )

```
String s = "This is a demo of the getChars method.";
int start = 10;
int end = 14;
char buf[] = new char[end - start];

s.getChars(start, end, buf, 0);

System.out.println(buf);
```

getBytes( )

stores the characters in an array of bytes.

byte[ ] getBytes( )

getBytes( )

```
String s = "ABCD";

byte b[] = s.getBytes();

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

toCharArray( )

to convert all the characters in a **String** object into a character array.
It returns an array of characters for the entire string.

char[ ] toCharArray( )

We can use getChars( ) to achieve the same result.

toCharArray( )

```
String s="mit manipal";
int count=0;

char c[] = s.toCharArray();
for(int i=0;i<c.length;i++)
    if(c[i] == ‘a')
            count++;
   System.out.println("count "+count);
```

regionMatches( )

compares a specific region inside a string with another specific region in another string.

general forms:

boolean regionMatches(int *startIndex*, String *str2*, int *str2StartIndex*, int *numChars*)

boolean regionMatches(boolean *ignoreCase*, int *startIndex*, String *str2*, int *str2StartIndex*, int *numChars*)

startIndex specifies the index at which the region begins within the invoking **String** object.
The **String** being compared is specified by *str2.*
The index at which the comparison will start within *str2* is specified by *str2StartIndex*.
The length of the substring being compared is passed in *numChars.*

In the second version, if *ignoreCase* is **true**, the case of the characters is ignored.

```
String s1 =  "Manipal Institute of Technology";
String s2 =  "National Institute of  Technology Karnataka";

 if(s1.regionMatches(8,s2,9,9))
        System.out.println("region matches");
 else
        System.out.println("region not matches");
```

```
String s1 =  "Manipal Institute of Technology";
String s2 =  "National Institute of  Technology Karnataka";

 if(s1.regionMatches(true,8,s2,9,9))
        System.out.println("region matches");
 else
        System.out.println("region not matches");
```

```
String s1 =  "Manipal Institute of Technology";
String s2 =  "National Institute of  Technology Karnataka";

 if(s1.regionMatches(false, 8,s2,9,9))
        System.out.println("region matches");
 else
        System.out.println("region not matches");
```

Count the number of words in a given string

Find the occurrences of each characters in a given string

StringBuffer

Constructors:

StringBuffer();

StringBuffer(int size);

StringBuffer(String s);

StringBuffer Methods

int capacity()

int length()

StringBuffer()

StringBuffer(int size)

StringBuffer(String s);

```
StringBuffer s1 = new StringBuffer();
 StringBuffer s2 = new StringBuffer(30);
 StringBuffer s3 = new StringBuffer("abcde");

  System.out.println(s1.capacity());
  System.out.println(s1.length());

    System.out.println(s2.capacity());
     System.out.println(s2.length());

    System.out.println(s3.capacity());
   System.out.println(s3.length());
```

16

0

30

0

21

5

StringBuffer Methods

char charAt(int pos);

void setCharAt(int i, char ch);

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

System.out.println("buffer before = " + sb);

sb.setCharAt(0, 'B');

System.out.println("buffer after = " + sb);
```

StringBuffer Methods

StringBuffer append(s) ;

insert( )

The **insert( )** method inserts one string into another.

StringBuffer insert(int *index*, String *str*)

StringBuffer insert(int *index*, char *ch*)

StringBuffer insert(int *index*, Object *obj*)

index specifies the index at which point the string will be inserted into the invoking **StringBuffer** object.

insert( )

```
StringBuffer sb = new StringBuffer(“Object Programming ");

sb.insert(7, “Oriented ");

System.out.println(sb);
```

reverse( )

reverse the characters within a **StringBuffer** object.

StringBuffer reverse( )

reverse( )

```
StringBuffer s = new StringBuffer(“malayalam");
System.out.println(“Given String is :”+s);
s.reverse();
System.out.println(“Reversed String is : “+s);
```

reverse( )

```
StringBuffer s = new StringBuffer(“liril");
System.out.println(“Given String is :”+s);
s.reverse();
System.out.println(“Reversed String is : “+s);
```

delete( ) and deleteCharAt( )

delete characters within a StringBuffer.

StringBuffer delete(int *startIndex*, int *endIndex*)

StringBuffer deleteCharAt(int *loc*)

startIndex : index of the first character to remove

endIndex : index one past the last character to remove.

substring deleted runs from *startIndex* to *endIndex*–1.

The **deleteCharAt(loc)** method deletes the character at the index specified by *loc.*

```
StringBuffer sb = new StringBuffer("Manipal Instititute of Technology");

        sb.delete(8, 24);
         System.out.println("After delete: " + sb);

         sb.deleteCharAt(0);
         System.out.println("After deleteCharAt: " + sb);
```

```
String s = “Object";
s.concat(“Oriented");
System.out.println(s);

StringBuffer s3 = new StringBuffer(“Object");
s3.append(“Oriented");
System.out.println(s3);
```

replace( )

replace one set of characters with another set inside a **StringBuffer** object.

StringBuffer replace(int *startIndex*, int *endIndex*, String *str*)

The substring being replaced is specified by the indexes *startIndex* and *endIndex.*

The substring at *startIndex* through *endIndex*–1 is replaced.

The replacement string is passed in *str.*

replace( )

```
StringBuffer sb = new StringBuffer("This is a test.");
sb.replace(5, 7, "was");
System.out.println("After replace: " + sb);
```

toString( )

Demonstrate String arrays

```
class StringDemo3
 {
        public static void main(String args[])
        {
                String str[] = { “one ”, “two”, “three” };
                for(int i =0; i<str.length; i++)
                System.out.println(str[i]);
        }
}
```

```
class prg
{
   public static void main(String args[])
   {
            StringBuffer s = new StringBuffer("mangalore");
            for(int i=0;i<s.length()-1;i++)
            for(int j=0;j<s.length()-1-i;j++)
            if(s.charAt(j) >s.charAt(j+1))
            {
                     char t = s.charAt(j);
                     s.setCharAt(j,s.charAt(j+1));
                     s.setCharAt(j+1,t);
            }
            System.out.println("sorted string : "+s);
   }
}
```

StringBuffer	StringBuilder
StringBuffer is synchronized i.e. thread safe.	StringBuilder is non-synchronized i.e. not thread safe.
StringBuffer is less efficient and slower than StringBuilder as StringBuffer is synchronized.	StringBuilder is more efficient and faster than StringBuffer.
StringBuffer is old, its there in JDK from very first release.	StringBuilder is introduced much later in release of JDK 1.5

Wrapper classes

Wrapper classes

- primitive types, such as **int** or **double** hold the basic data types supported by the language.

- Many of the standard data structures implemented by Java operate on objects.

- Java provides *type wrappers*, which are classes that encapsulate a primitive type within an object.

Wrapper classes

- The type wrappers are **Double**, **Float**, **Long**, **Integer**, **Short**, **Byte**, **Character**, and **Boolean**.

- Offer many methods that allow us to fully integrate the primitive types into Java's object hierarchy.

Wrapper classes

Primitive datatypes can be converted into object types by using wrapper classes .

Simple Type	Wrapper class
boolean	Boolean
char	Character
double	Double
float	Float
int	Integer
long	Long

Wrapper classes

Character

Character is a wrapper around a char.

Character(char *ch*)

Here, *ch* specifies the character that will be wrapped by the **Character** object being created.

To obtain the **char** value contained in a **Character** object :

char charValue( )

Wrapper classes

Boolean

Boolean is a wrapper around boolean values.

Boolean(boolean *boolValue*)

Boolean(String *boolString*)

In the first version, *boolValue* must be either **true** or **false**.

In the second version, if *boolString* contains the string “true” (in uppercase or lowercase), then the new **Boolean** object will be true. Otherwise, it will be false.

To obtain a **boolean** value from a **Boolean** object :

boolean booleanValue( )

Wrapper classes

The Numeric Type Wrappers

The most commonly used type wrappers are those that represent numeric values.

These are **Byte**, **Short**, **Integer**, **Long**, **Float**, and **Double**.

All of the numeric type wrappers inherit the abstract class **Number**.

Number declares methods that return the value of an object in each of the different number formats. These methods are :

byte byteValue( )

short shortValue( )

int intValue( )

int longValue()

float floatValue()

double doubleValue( )

These methods are implemented by each of the numeric type wrappers.

Wrapper classes

- All of the numeric type wrappers define constructors that allow an object to be constructed from a given value, or a string representation of that value.

- the constructors defined for **Integer**:

  Integer(int *num*)

  Integer(String *str*)

- If *str* does not contain a valid numeric value, then a **NumberFormatException** is thrown.

Wrapper classes

```
// Demonstrate a type wrapper.
class Wrap
{
        public static void main(String args[])
        {
                Integer iOb = new Integer(100);

                int i = iOb.intValue();

                System.out.println(i + " " + iOb); // displays 100 100
         }
}
```

Wrapper classes

boxing : The process of encapsulating a value within an object.

Below line boxes the value 100 into an **Integer**:

```
Integer iOb = new Integer(100);
```

unboxing : The process of extracting a value from a type wrapper.

Below line unboxes the value in **iOb.**

```
int i = iOb.intValue();
```

Wrapper classes

autoboxing and auto-unboxing

- Beginning with JDK 5, Java added two important features: *autoboxing* and *auto-unboxing.*

- Autoboxing : the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper.
- There is no need to explicitly construct an object.

- Auto-unboxing : the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed.

- There is no need to call a method such as intValue( ) or doubleValue( ).

Wrapper classes

- autoboxing and auto-unboxing streamlines the coding of several algorithms, removing the tedium of manually boxing and unboxing values.
- It also helps prevent errors.
- It is very important to generics, which operates only on objects.
- autoboxing makes working with the Collections Framework much easier.

Wrapper classes

- With autoboxing it is no longer necessary to manually construct an object in order to wrap a primitive type.
- We need only assign that value to a type-wrapper reference.
- Java automatically constructs the object.

Below way to construct an **Integer** object that has the value 100:

```
Integer iOb = 100; // autobox an int
```

Note : no object is explicitly created through the use of **new**. Java handles this automatically.

Wrapper classes

To unbox an object, simply assign that object reference to a primitive-type variable.

to unbox **iOb**, we can use this line:

int i = iOb; // auto-unbox

Wrapper classes

```
 // Demonstrate autoboxing/unboxing.
class AutoBox
{
	public static void main(String args[])
	{
		Integer iOb = 100;    // autobox an int
		int i = iOb;    // auto-unbox
		System.out.println(i + " " + iOb); // displays 100 100
	}
}
```

Wrapper classes

autoboxing and methods

- In addition to the simple case of assignments, autoboxing automatically occurs whenever a primitive type must be converted into an object;
- auto-unboxing takes place whenever an object must be converted into a primitive type.
- autoboxing/unboxing might occur when an **argument is passed to a method**, or when a value is returned by a method.

Wrapper classes

```
 class AutoBox2
{
        static int m(Integer v)
        {       return v ; // auto-unbox to int
        }

        public static void main(String args[])
        {
              Integer iOb = m(100);
             System.out.println(iOb);
         }
}
```

Wrapper classes

- Here **m( )** specifies an **Integer** parameter and returns an **int** result.
- Inside **main( )**, **m( )** is passed the value 100. Because **m( )** is expecting an **Integer**, this value is automatically boxed.
- Then, **m( )** returns the **int** equivalent of its argument.
- This causes **v** to be auto-unboxed.
- Next, this **int** value is assigned to **iOb** in **main( )**, which causes the **int** return value to be autoboxed.

Wrapper classes

Autoboxing/Unboxing Occurs in Expressions

- autoboxing and unboxing also applies to expressions.
- Within an expression, a numeric object is automatically unboxed.
- The outcome of the expression is reboxed, if necessary.

Wrapper classes

```
class AutoBox3
{       public static void main(String args[])
        {         Integer iOb, iOb2;
                  int i;
                  iOb = 100;
                 System.out.println("Original value of iOb: " + iOb);
                  ++iOb;
                 System.out.println("After ++iOb: " + iOb);
                 iOb2 = iOb + (iOb / 3);
                System.out.println("iOb2 after expression: " + iOb2);
                i = iOb + (iOb / 3);
                System.out.println("i after expression: " + i);
   }}
```

Wrapper classes

++iOb;

This causes the value in **iOb** to be incremented.

Here **iOb** is unboxed, the value is incremented, and the result is reboxed.

Wrapper classes

Auto-unboxing allows to mix different types of numeric objects in an expression.

Once the values are unboxed, the standard type conversions are applied.

The result is reboxed and stored in **dOb**.

```
class AutoBox4
 {      public static void main(String args[])
     {          Integer iOb = 100;
                Double dOb = 98.6;
                dOb = dOb + iOb;
                System.out.println("dOb after expression: " + dOb);
     }
}
```

Wrapper classes

Because of auto-unboxing, we can use integer numeric objects to control a **switch** statement.

For example, consider this fragment:

```
Integer iOb = 2;
switch(iOb)
{
     case 1: System.out.println("one");
                 break;
    case 2: System.out.println("two");
                 break;
     default: System.out.println("error");
}
```

Wrapper classes

Autoboxing/Unboxing Boolean and Character Values

```
class AutoBox5
{    public static void main(String args[])
     {
          // Autobox/unbox a boolean.
          Boolean b = true;
          // b is auto-unboxed.
          if(b) System.out.println("b is true");

           // Autobox/unbox a char.
          Character ch = 'x'; // box a char
           char ch2 = ch; // unbox a char
           System.out.println("ch2 is " + ch2);
}}
```

Wrapper classes

- Because of auto-unboxing, a **Boolean** object can now also be used to control loop statements.
- When a **Boolean** is used as the conditional expression of a **while**, **for**, or **do/while**, it is automatically unboxed into its **boolean** equivalent.

```
Boolean b;
// ...
while(b) { // ... }
```

Method calling	Conversion action
st = Integer.toString(i);	primitive integer to string.
st = Float.toString(f);	primitive float to string.
st = Double.toString(d);	primitive double to string.
st = Long.toString(l);	primitive long to string.

method calling	**conversion action**
int i = Integer.parseInt(st)	Converts string to primitive integer
long l = Long.parseLong(st)	Converts string to primitive long
Double d = Double.parseDouble(st)	Converts string to primitive double
Float f = Float.parseFloat(st)	Converts string to primitive float

Method calling	Conversion action
obj = Double.valueOf(st)	Converting string to double object
obj = Float.valueOf(st)	Converting string to float object
obj = Integer.valueOf(st)	Converting string to integer object
obj = Long.valueOf(st)	Converting string to long object

```
class Mark
{       int sessionalMark, finalMark;
         void setMark(int a,int b)
          {
                sessionalMark =a;
                finalMark = b;
          }

         void dispMark()
         {
             System.out.println("sessional mark : "+sessionalMark);
             System.out.println("final mark : "+finalMark);
        }
}
```

```
class Student
{
        int regno;
        String name;
        Mark m;

        Student(int a, String b,int c, int d)
        {
                regno = a;
                name = b;
                m = new Mark();
                m.setMark(c,d);
        }
        void disp()
        {       System.out.println("Regno : "+regno);
                System.out.println("name : "+name);
                m.dispMark();
        } }
```

```
class PrgObjectMember
{
        public static void main(String args[])
        {
                Student s1 = new Student(1,"anil",10,20);
                Student s2 = new Student(2,"sunil",30,40);
                Student s3 = new Student(3,"rahul",50,60);
                s1.disp();
                s2.disp();
                s3.disp();
        }
}
```

```
class Student
{
        int regno;
        String name;

       void Count()
       {
              System.out.println("number of objects "+cc);
       }
}
class PrgObjectMember
{
        public static void main(String args[])
        {
              Student s1 = new Student(1,"anil");
              Student s2 = new Student(2,"sunil");
              Student s3 = new Student(3,"rahul");
        }
}
```

```
class Student
{
        int regno;
        String name;
    Student(int a, String b,int c, int d)
        {
                regno = a;
                name = b;
        }
        void disp()
        {
                System.out.println("Regno : "+regno);
                System.out.println("name : "+name);
        }

}
```

```
class ObjectInsert
{
        public static void main(String args[])
        {
                Student s[] = new Student[10];
                 s[0] = new Student(1,"anil",10,20);
                 s[1]= new Student(2,"sunil",30,40);
                 s[2] = new Student(3,"rahul",50,60);
                for(int i=0;i<3;i++)
                s[i].disp();
               insert(s,3);
              System.out.println("after insertion");
               for(int i=0;i<4;i++)
               s[i].disp();
          }
        static void insert(Student s1[],int l)
        {
        }
}
```

PresentationPoint