# Laboratory Manual

**Subject: - Programming Laboratory-I (Java)**

**Class:-SY BTech** Computer Engineering **Semester-**III

**Experiment No:-11**

**Aim: -** Program on String Buffer.

**Software requirement**: - Java

**Theory: -**

**StringBuffer** is a peer class of **String** that provides much of the functionality of strings. The string represents fixed-length, immutable character sequences while StringBuffer represents growable and writable character sequences. **StringBuffer** may have characters and substrings inserted in the middle or appended to the end. It will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth.

**StringBuffer class** is used to create mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

**Important Constructors of StringBuffer class**

- StringBuffer(): creates an empty string buffer with the initial capacity of 16.
- StringBuffer(String str): creates a string buffer with the specified string.
- StringBuffer(int capacity): creates an empty string buffer with the specified capacity as length.

| Methods | Action Performed |
|---|---|
| append() | Used to add text at the end of the existing text. |
| length() | The length of a StringBuffer can be found by the length( ) method |
| capacity() | the total allocated capacity can be found by the capacity( ) method |
| charAt() | This method returns the char value in this sequence at the specified index. |
| delete() | Deletes a sequence of characters from the invoking object |
| deleteCharAt() | Deletes the character at the index specified by *loc* |
| ensureCapacity() | Ensures capacity is at least equals to the given minimum. |
| insert() | Inserts text at the specified index position |
| length() | Returns length of the string |
| reverse() | Reverse the characters within a StringBuffer object |
| replace() | Replace one set of characters with another set inside a StringBuffer object |

```java
//Program on StringBuffer

public class StringBufferExample {

    public static void main(String[] args) {
    StringBuffer sb = new StringBuffer("Hello");

    sb.append(" java");
    System.out.println(sb);
    sb.insert(1,"programming");
    System.out.println(sb);
    sb.replace(0,4,"Eclipse");
    System.out.println(sb);
    sb.delete(4,7);
    System.out.println(sb);
    sb.reverse();
    System.out.println(sb);
    sb.append("java is my favourite language");
    System.out.println(sb.capacity());
    }
}
```

# Laboratory Manual

**Subject: - Programming Laboratory-I (Java)**

**Class:-SY BTech** Computer Engineering **Semester-**III

**Experiment No:-12**

**Aim: -** Program on Vector.

**Software requirement**: - Java

**Theory: -** The Vector class synchronizes each individual operation. This means whenever we want to perform some operation on vectors, the Vector class automatically applies a lock to that operation.

It is because when one thread is accessing a vector, and at the same time another thread tries to access it, an exception called ConcurrentModificationException is generated. Hence, this continuous use of lock for each operation makes vectors less efficient.

**Creating a Vector:**

**Vector<Type> vector = new Vector< >( );**

….. Type indicates the type of a linked list.

For example:

```
// create Integer type linked list
Vector<Integer> vector= new Vector< >( );

// create String type linked list
Vector<String> vector= new Vector< >( );
```

**Add Elements to Vector:**

1. add(element) - adds an element to vectors
2. add(index, element) - adds an element to the specified position
3. addAll(vector) - adds all elements of a vector to another vector

**Access Vector Elements:**

1. get(index) - returns an element specified by the index
2. iterator() - returns an iterator object to sequentially access vector elements

**Remove Vector Elements:**

1. remove(index) - removes an element from specified position
2. removeAll() - removes all the elements
3. clear() - removes all elements. It is more efficient than removeAll()

```
// Program: Java vector

import java.util.*;
public class VectorExample {

        public static void main(String[] args) {
          //create a vector
          Vector<String> vec = new Vector<String>();
          //Adding elements using add() method of list
          vec.add("Tiger");
          vec.add("Lion");
          vec.add("Dog");
          vec.add("Elephant");
          //Adding elements using addElement() method of vector
          vec.addElement("Rat");
          vec.addElement("Cat");
          vec.addElement("Deer");
          System.out.println("Elements are : "+vec);
        }

}
```

# Laboratory Manual

**Subject: - Programming Laboratory-I (Java)**

**Class:-SY BTech** Computer Engineering **Semester-**III

**Experiment No:-13**

**Aim: -** Program on single and multilevel inheritance (Use super keyword).

**Software requirement**: - Java

**Theory: - Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object. The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

**Types of inheritance**

Java supports the following four types of inheritance:

1. Single Inheritance
2. Multi-level Inheritance
3. Hierarchical Inheritance
4. Hybrid Inheritance

**Single Inheritance:**

- In single inheritance, a sub-class is derived from only one super class. It inherits the properties and behavior of a single-parent class. Sometimes it is also known as simple inheritance**.**

```
public class Employee
{
 float salary=40000;
}

class Programmer extends Employee
{
     int bonus=10000;
          public static void main(String args[])
          {
            Programmer p=new Programmer();
            System.out.println("Programmer salary is:"+p.salary);
            System.out.println("Bonus of Programmer is:"+p.bonus);
          }
}
```

**Multi-level Inheritance:**

In multi-level inheritance, a class is derived from a class which is also derived from another class is called multi-level inheritance. In simple words, we can say that a class that has more than one parent class is called multi-level inheritance. Note that the classes must be at different levels. Hence, there exists a single base class and single derived class but multiple intermediate base classes

```
class Student
{
    int reg_no;

    Student()
    {
        System.out.println("Welcome to rcpit");
    }
    void getNo(int no)
    {
        reg_no=no;
    }

    void putNo()
    {
        System.out.println("registration No.= "+reg_no);
    }


}
```

```
class Marks extends Student
{
    Marks()
    {
        super();
    }
    float marks;

    void getMarks(float m)
    {
        marks=m;
    }
    void putMarks()
    {
        System.out.println("marks= "+marks);
    }
}
```

```
class Sports extends Marks
{
    float score;
    void getScore(float scr)
    {
        score=scr;
    }
    void putScore()
    {
        System.out.println("score= "+score);
    }
}

public class Test
{
    public static void main(String args[])
    {
        Sports ob=new Sports();
        ob.getNo(987);
        ob.putNo();
        ob.getMarks(78);
        ob.putMarks();
        ob.getScore(68.7f);
        ob.putMarks();
    }
}
```

# Laboratory Manual

**Subject: - Programming Laboratory-I (Java)**

**Class:-SY BTech** Computer Engineering **Semester-**III

**Experiment No:-14**

**Aim: -** Program on abstract class.

**Software requirement**: - Java

**Theory: -** 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).

It needs to be extended and its method implemented. It cannot be instantiated.

Points to Remember

1. An abstract class must be declared with an abstract keyword.
2. It can have abstract and non-abstract methods.
3. It cannot be instantiated.
4. It can have constructors and static methods also.
5. It can have final methods which will force the subclass not to change the body of the method.

**//Program for abstract class**

```
abstract class shape
{
public int x, y;
public abstract void printArea();
}
class Rectangle extends shape
{
public void printArea()
{
System.out.println("Area of Rectangle is " + x * y);
}
}
class Triangle extends shape
{
public void printArea()
{
System.out.println("Area of Triangle is " + (x * y) / 2);
}
}
class Circle extends shape
{
public void printArea()
{
System.out.println("Area of Circle is " + (22 * x * x) / 7);
}
}
public class Abstex
{
public static void main(String[] args)
{
Rectangle r = new Rectangle();
r.x = 10;
r.y = 20;
r.printArea();
System.out.println("-----------------------------------");
Triangle t = new Triangle();
t.x = 30;
t.y = 35;
t.printArea();
System.out.println("-----------------------------------");
Circle c = new Circle();
c.x = 2;
c.printArea();
System.out.println("-----------------------------------");
}
}
```

# Laboratory Manual

**Subject: - Programming Laboratory-I (Java)**

**Class:-SY BTech** Computer Engineering **Semester-**III

**Experiment No:-15**

**Aim: -** Program on interface demonstrating concept of multiple inheritance.

**Software requirement**: - Java

**Theory:**

Java is an **Object Oriented Programming language** and supports the feature of inheritance. We cannot have Multiple Inheritance in Java directly due to Diamond Problem but it can be implemented using Interfaces.

Interface is a collection of abstract methods (non-defined methods). A class implements an interface, hence inheriting the abstract methods of the interface. An interface may also contain constants, Variables, default methods, static blocks, methods etc.

We can achieved *Multiple inheritance* in java with the help of interfaces where we have declared our methods in the interfaces and overridden them in the child class. (*Override* means when a method that has already been declared in parent class is being defined again in the child class having same name, number and type of parameters, and return type as the method that it overrides) .

```java
//interface 1
interface ParentA
{
    //interfaces method are only declared not defined
    public void walk();
}

//interface 2
interface ParentB
{
    //any number of methods can be declared in the interface
    public void walk();
    public void run();
}

//multiple inheritance achieved
class child implements ParentA,ParentB
{
    //overidden methods
    public void walk()
    {
        System.out.println("ParentA is walking ");
    }
    public void run()
    {
        System.out.println("ParentB is running ");
    }
    public static void main (String args[])
    {
        child object = new child();
        object.walk();
        object.run();
    }
}
```

# Laboratory Manual

**Subject: - Programming Laboratory-I (Java)**

**Class:-SY BTech** Computer Engineering **Semester-**III

**Experiment No:-16**

**Aim: -** Program on dynamic method dispatch using base class and interface reference.

**Software requirement**: - Java

**Theory:**

**Dynamic method dispatch** is the mechanism in which a call to an overridden method is resolved at run time instead of compile time. This is an important concept because of how Java implements run-time polymorphism.

Java uses the principle of *'a superclass reference variable can refer to a subclass object'* to resolve calls to overridden methods at run time. When a superclass reference is used to call an overridden method, *Java determines which version of the method* to execute based on the type of the object being referred to at the time call. In other words, it is the type of object being referred to that determines which version of an overridden method will be executed.

```java
interface Mobile
{

    void showTime();
    void  on();

}
class Phone implements Mobile{
    public void showTime(){
        System.out.println("Time is 8 am");
    }
    public void on(){
        System.out.println("Turning on Phone...");
    }
}

class SmartPhone extends Phone{
    public void music(){
        System.out.println("Playing music...");
    }
    public void on(){
        System.out.println("Turning on SmartPhone...");
    }
}
public class CWH {
    public static void main(String[] args) {

        Phone obj = new SmartPhone();


        obj.showTime();
        obj.on();
    }
}
```