MUFFAKHAM JAH COLLEGE OF ENGINEERING AND TECHNOLOGY

Banjara Hills, Hyderabad, Telangana


DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

# JAVA Laboratory Manual


Academic Year 2016-2017

# Table of Contents

## I Contents



## II Programs

# Part I

# Contents

# 1. Vision of the Institution

To be part of universal human quest for development and progress by contributing high calibre, ethical and socially responsible engineers who meet the global challenge of building modern society in harmony with nature.

# 2. Mission of the Institution

- To attain excellence in imparting technical education from undergraduate through doctorate levels by adopting coherent and judiciously coordinated curricular and co-curricular programs.
- To foster partnership with industry and government agencies through collaborative research and consultancy.
- To nurture and strengthen auxiliary soft skills for overall development and improved employability in a multi-cultural work space.
- To develop scientific temper and spirit of enquiry in order to harness the latent innovative talents.
- To develop constructive attitude in students towards the task of nation building and empower them to become future leaders
- To nourish the entrepreneurial instincts of the students and hone their business acumen.
- To involve the students and the faculty in solving local community problems through economical and sustainable solutions.

## 3. Department Vision

To contribute competent computer science professionals to the global talent pool to meet the constantly evolving societal needs.

## 4. Department Mission

Mentoring students towards a successful professional career in a global environment through quality education and soft skills in order to meet the evolving societal needs.

# 5. Programme Education Objectives

1. Graduates will demonstrate technical skills and leadership in their chosen fields of employment by solving real time problems using current techniques and tools.

2. Graduates will communicate effectively as individuals or team members and be successful in the local and global cross cultural working environment.

3. Graduates will demonstrate lifelong learning through continuing education and professional development.

4. Graduates will be successful in providing viable and sustainable solutions within societal, professional, environmental and ethical contexts

# 6. Programme Outcomes

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Lifelong learning:** Recognize the need for, and have the preparation and ability to engage in independent and lifelong learning in the broadest context of technological change.

# 7. Programme Specific Outcomes

The graduates will be able to:

**PSO1:** Demonstrate understanding of the principles and working of the hardware and software aspects of computer systems.

**PSO2:** Use professional engineering practices, strategies and tactics for the development, operation and maintenance of software

**PSO3:** Provide effective and efficient real time solutions using acquired knowledge in various domains.

# 8. Introduction To Java Laboratory

Introduction to both a programming language and a platform

A class is the collection of related data and function under a single name. A C++ program can have any number of classes. When related data and functions are kept under a class, it helps to visualize the complex problem efficiently and effectively.

**The Java Programming Language**

The Java programming language is a high-level language that can be characterized by all of the following buzzwords:

- Simple
- Architecture neutral
- Portable
- Object oriented
- Distributed
- High performance
- Robust
- Multi threaded
- Dynamic
- Secure

In the Java programming language, all source code is first written in plain text files ending with the .java extension. Those source files are then compiled into .class files by the javac compiler. A .class file does not contain code that is native to a processor; it instead contains bytecodes — the machine language of the Java Virtual Machine (Java VM). The java launcher tool then runs your application with an instance of the Java Virtual Machine.

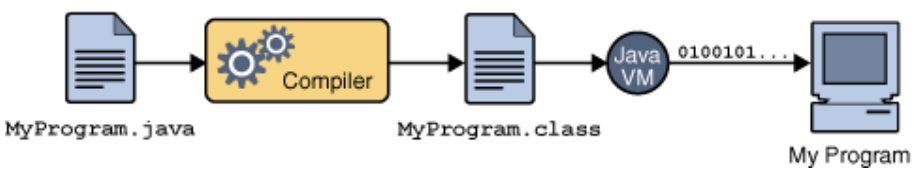



Fig. 1: An overview of the software development process.

Because the Java VM is available on many different operating systems, the same .class files are capable of running on Microsoft Windows, the Solaris Operating System (Solaris OS), Linux, or Mac OS. Some virtual machines, such as the "Java HotSpot virtual machine", perform additional steps at runtime to give java application a performance boost. This includes various tasks such as finding performance bottlenecks and recompiling (to native code) frequently used sections of code.

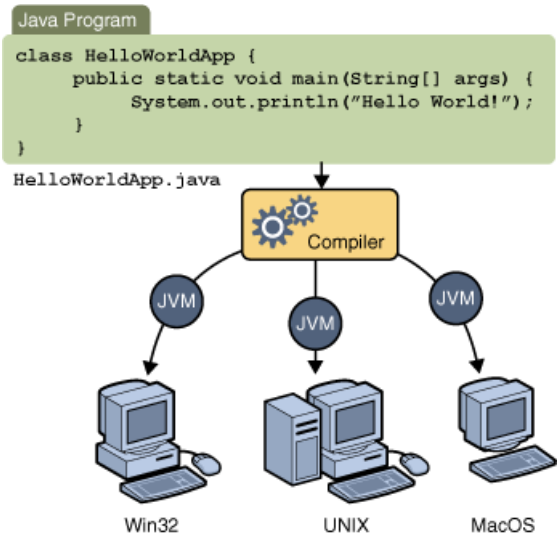



Fig. 2: Java VM, on multiple platforms.

Through The Java VM, the same application is capable of running on multiple platforms.

### The Java Platform

A platform is the hardware or software environment in which a program runs (some example popular platforms like Microsoft Windows, Linux, Solaris OS, and Mac OS). Most platforms can be described as a combination of the operating system and underlying hardware. The Java platform differs from most other platforms in that it's a software-only platform that runs on top of other hardware-based platforms.

The Java platform has two components:

- The Java Virtual Machine
- The Java Application Programming Interface (API)

**The Java Virtual Machine:**

It's the base for the Java platform and is ported onto various hardware-based platforms.The API is a large collection of ready-made software components that provide many useful capabilities. It is grouped into libraries of related classes and interfaces; these libraries are known as packages.

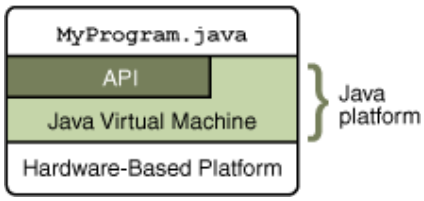



**Fig. 3: JVM as platform**

The API and Java Virtual Machine insulate the program from the underlying hardware. As a platform-independent environment, the Java platform can be a bit slower than native code. However, advances in compiler and virtual machine technologies are bringing performance close to that of native code without threatening portability.

**Versions**

- JDK 1.0 1996, Solaris, Windows, MacOS Classic, Linux
- JDK 1.1 1997, Solaris, Windows, MacOS Classic, Linux
- JDK 1.2 1998 (also known as Java 2), Solaris, Windows, Linux
- JDK 1.3 2000, Solaris, Windows, MacOS X, Linux
- JDK 1.4 2002, Solaris, Windows, MacOS X, Linux
- JDK 1.5 2004 Solaris, Windows, MacOS X, Linux
- JDK 1.6 2006 Solaris, Windows, MacOS X, Linux
- JDK 1.7 2011 Solaris, Windows, MacOS X, Linux
- JDK 1.8 2014 Solaris, Windows, MacOS X, Linux

**Procedure for Compiling and Executing Java Applications Compiling and Executing Console I/O Applications:-**

1. Create a file XXXX.java using vi editor where, XXXX can be any filename and .java is the extension used for the java source files.
2. Compile it as javac XXXX.java ( For compilation specify the file-name)
3. Execute the program as java Class-name (Class-name that contains main( ) method)

**Compiling and Executing Java Applets:-**

1. Create a file XXXX.java using notepad in windows operating system.
2. XXXX can be any filename and .java is the extension used for the java source files.
3. Applet class must be declared as public. Therefore, Applet class-name must match exactly with filename.
4. Write the applet tag code in java source file using java multi-line comments.
   For example,

```
/*
<applet  code = "StudentForm"   width = 422     height = 400 >
</applet>
*/
```

5. Compile it as javac XXXX.java (Filename).
6. Execute the program using appletviewer tool as, appletviewer Filename.java.

**Data Types**

Java is a strongly typed language, since

1. Every variable has a type, every expression has a type, and every type is strictly defined.
2. All assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility.
3. There are no automatic coercions or conversions of conflicting types.
4. The Java compiler checks all expressions and parameters to ensure that the types are compatible. Any type mismatches are errors.

### The Primitive Types

Java defines eight primitive types of data: byte, short, int, long, char, float, double, and boolean.

| S. No. | Data types | Size (in bits) | Range | |
|---|---|---|---|---|
| | | | Min Value | Max Value |
| 1. | byte | 8 | $-2^7$ | $2^7-1$ |
| 2. | short | 16 | $-2^{15}$ | $2^{15}-1$ |
| 3. | int | 32 | $-2^{31}$ | $2^{31}-1$ |
| 4. | long | 64 | $-2^{63}$ | $2^{63}-1$ |
| 5. | float | 32 | 1.4e-45 | 3.4e+38 |
| 6. | double | 64 | 4.9e-324 | 1.8e+308 |
| 7. | char (Unicode) | 16 | 0 | 65535 |
| 8. | boolean | Not precisely defined | true/false | |

**Table 1: Primitive Data types in Java**

### Literals

A literal is the source code representation of a fixed value; literals are represented directly in your code without requiring computation. Primitive types are special data types built into the language; they are not objects created from a class. As shown below, it's possible to assign a literal to a variable of a primitive type:

```
boolean result = true;
    char capitalC = 'C';
    byte b = 100;
    short s = 10000;
    int i = 100000;
```

The integral types (byte, short, int, and long) can be expressed using decimal, octal, or hexadecimal number systems. Decimal is the number system you already use every day; it's based on 10 digits, numbered 0 through 9. The octal number system is base 8,consisting of the digits 0 through 7. The hexadecimal system is base 16, whose digits are the numbers 0 through 9 and the letters A through F. For general-purpose programming, the decimal system is likely to be the only number system you'll ever use. However, if you need octal or hexadecimal, the following example shows the correct syntax. The prefix 0 indicates octal, whereas 0x indicates hexadecimal.

```
int decVal = 26;  // The number 26, in decimal
int octVal = 032;  // The number 26, in octal
int hexVal = 0x1a; // The number 26, in hexadecimal
```

The floating point types (float and double) can also be expressed using E or e (for scientific notation), F or f (32-bit float literal) and D or d (64-bit double literal; this is the default and by convention is omitted).

```
  double d1 = 123.4;
    double d2 = 1.234e2;
// same value as d1, but in scientific notation
    float f1  = 123.4f;
```

Literals of types char and String may contain any Unicode (UTF-16) characters. If your editor and file system allow it, you can use such characters directly in your code. If not, you can use a "Unicode escape" such as ' u0108' (capital C with circumflex), or "Sŏ0ED seŏ0F1or"(Sí Señor in Spanish). Always use 'single quotes' for char literals and "double quotes" for String literals. Unicode escape sequences may be used elsewhere in a program (such as in field names, for example), not just in char or String literals.

The Java programming language also supports a few special escape sequences for char and String literals:

```
\b (backspace)
\t (tab)
\n (line feed)
\f (form feed)
\r (carriage return)
\" (double quote)
\' (single quote)
\\ (backslash)
```

There's also a special null literal that can be used as a value for any reference type.null may be assigned to any variable, except variables of primitive types.

**Type Casting**

***Implicit Type Casting:***

Java performs automatic type casting when the following two conditions are met :

- The types are compatible.
- The destination type is larger than the source type.

This is also called widening conversion.

***Explicit Type Casting:***

To perform a conversion between two incompatible types, we must explicitly use a cast operation. The general form is:

```
(target-type)value
```

For example, to store an int value into a byte variable or storing a float value into int variable. This type of conversion is sometimes also referred as narrowing conversion.Explicit typecasting must be done carefully as it may also result in improper conversions.

```
int p =  100;
byte b = (byte) p ;
    // p value is within byte range, therefore b gets 100.
byte b = (byte) 257;
   // b gets the value 1 i.e., equal to  257 % 256(byte's range).
int balance = (int) 324.75;
   //results in truncation , and balance gets 324.
```

## Concept of Class

A class is a template for an object, and an object is an instance of a class.

The General Form of a Class

- A class is declared by using the keyword class. A class declaration only creates a template; it does not create an actual object.

```
class classname {

type  instance-variable1;
type  instance-variable2;
// ...
type  instance-variableN;

   type method-name1(parameter-list)
{

// body of method
}
type method-name2(parameter-list)
{

// body of method
}

// ...
type method-nameN(parameter-list)
{

// body of method
}
}
```

1. Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables.
2. The code is contained within methods.
3. Collectively, the methods and variables defined within a class are called members of the class.

```
For example,
class Box {

double width;
double height;
double depth;
}
```

The Box class is a simple class that contains only three data members i.e., width, height and depth. These data members of the class are also referred as instance variables (or attributes) of a class.

- A class defines a new data type. In this case, the new data type is called Box.
- We can use this name to declare objects (objects of type Box).

**Objects:**

Object is defined as an instance of a class.

A Box object can be created as,

```
Box mybox = new Box( ); // create a Box object called mybox
```

Actually obtaining objects of a class is a two-step process:

1. Declare a variable of the class type called a reference variable.
2. Acquire a physical (or actual) copy of the object by using the new operator.

**Reference Variable**

This variable does not define an object. Instead, it is simply a variable that can refer to an object.

### Operator new

The new operator dynamically allocates memory (that is, allocates at run time) for an object and returns a reference to it.

So instead of saying,

```
Box mybox = new Box( );
```

It can be written more clearly as:

```
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```

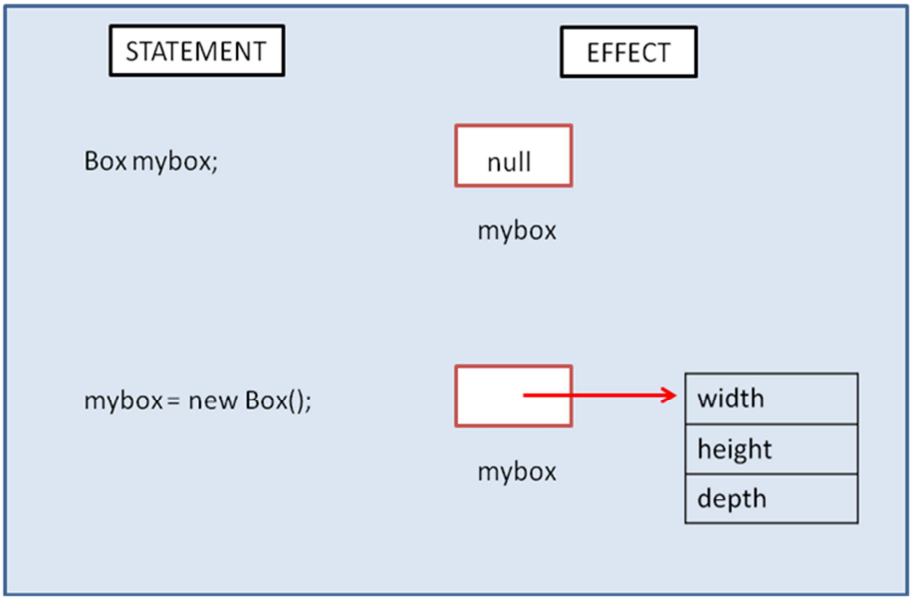



**Fig. 4: Declaring an Object of type Box**

### Accessing instance variables of Box object

Instance variables of an object can be accessed using the dot (.) operator.

General Form,

```
Object-reference . instance_var-name
                Example,
                       mybox.width = 100;
                       mybox.height = 50;
                       mybox.depth = 25;
```

**File Name:BoxDemo1.java**

**/* Simple Java file to demonstrate creation of class and object*/**

```
class Box {

double width;
double height;
double depth;
}
class BoxDemo1 {

public static void main(String args[ ]) {

Box mybox = new Box( ); // creates Object of Box class.
double vol;

// assign values to mybox's instance variables
mybox.width = 3;
mybox.height = 2;
mybox.depth = 1;

// compute volume of box
vol = mybox.width * mybox.height * mybox.depth;
System.out.println("Volume is " + vol);
}
}
```

Every object contains its own copy of instance variables defined in the class. i.e., every Box object will contain its own copies of the instance variables width, height,and depth.

**Program 2**

**File Name: BoxDemo2.java**

**Program to illustrate each object has its own copies of the instance variables.**

**Problem Definition:-**

/*Program to illustrate each object has its own copies of the instance variables.*/

**Problem Description: -**

/*Program to illustrate each object has its own copies of the instance variables.*/

```
class Box {
double width;
double height;
double depth;
}
class BoxDemo2 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
// assign values to mybox2's instance variables
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
// compute volume of first box
vol = mybox1.width * mybox1.height * mybox1.depth;
System.out.println("Volume is " + vol);
// compute volume of second box
vol = mybox2.width * mybox2.height * mybox2.depth;
System.out.println("Volume is " + vol);
}
}
```

**Introducing Methods**

The general form of a method is:

```
type method-name(parameter-list) {

// body of method
}
```

1. type specifies the type of data returned by the method.
2. The name of the method is specified by method-name.
3. The parameter-list is a sequence of type and identifier pairs separated by commas.

Adding a Method to the Box Class

Instead of finding up the volume in other class by accessing the data members width, height and depth, we can add a method called volume( ) in Box class itself.

**File Name: BoxDemo3.java**
**/* Program to demonstrate how methods can be written in a class. */**
**/* Method volume( ) is added to Box class.*/**

```
   class Box {
double width;
double height;
double depth;
double volume( ) { // Method to compute and return volume of box.
return width * height * depth;
}
}
class BoxDemo3 {
public static void main(String args[ ]) {
Box mybox1 = new Box( );
Box mybox2 = new Box( );
double vol;
// assign values to mybox1's instance variables.
mybox1.width = 3;
mybox1.height = 2;
mybox1.depth = 1;
// assign different values to mybox2's instance variables.
mybox2.width = 4;
mybox2.height = 2;
mybox2.depth = 1;
// get volume of first box
vol = mybox1.volume( );
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume( );
System.out.println("Volume is " + vol);
}
}
```

### Adding a Method That Takes Parameters

Instead of assigning values to width, height and depth instance variables in other class, we can add a

```
setDim(int w, int h, int d)
```

in Box class itself.

**File Name: BoxDemo4.java**
**Java program to illustrate how parameterized method can be added in a class.**

```
class Box {
double width;
double height;
double depth;
double volume( ) {
return width * height * depth;
}
void setDim(double w, double h, double d) { // set dimensions of the box.
width = w;
height = h;
depth = d;
}
}
class BoxDemo4 {
public static void main(String args[ ]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// assign mybox1's width to 1, height to 2 and depth to 3.
mybox1.setDim(1, 2, 3);
// assign mybox2's width to 4, height to 2 and depth to 1.
mybox2.setDim(4, 2, 1);
vol = mybox1.volume( );  // gets volume of mybox1.
System.out.println("Volume is " + vol);
vol = mybox2.volume();  // gets volume of mybox2.
System.out.println("Volume is " + vol);
}
}
```

**Constructors:**
A constructor initializes an object immediately upon creation.

1. It has the same name as the class in which it resides and is syntactically similar to a method.
2. Once defined, the constructor is automatically called immediately after the object is created.
3. Constructors do not have any return type, not even void.

Constructor's ensures an object is properly initialized, before it is used.

**File Name:BoxDemo5.java**

**/*Java Program to illustrate how a constructor is specified in a class*/**

```
class Box {
double width;
double height;
double depth;
Box( ) { // No-argument Constructor of Box class.
width = 10;
height = 5;
depth = 1;
}
double volume( ) {
return width * height * depth;
}
}
class BoxDemo5 {
public static void main(String args[ ]) {
// creating two Box objects.
Box mybox1 = new Box( );
Box mybox2 = new Box( );
double vol; // To store volume of box.
vol = mybox1.volume( ); // get volume of first box
System.out.println("Volume is " + vol);
vol = mybox2.volume( );  // get volume of second box
System.out.println("Volume is " + vol);
}
}
```

### Parameterized Constructors

Previous version of Box class constructor is not very useful—as it initializes all objects of Box class with the same dimensions (i.e., with width=10 , height = 5 and depth = 1). Parameterized constructors can be used to create box objects with customized values for width, height and depth.

**File Name: BoxDemo6.java**

**Java Program that demonstrate how a parameterized constructor is used toinitialize the dimensions of a box..**

```
class Box {
double width;
double height;
double depth;
Box(double w, double h, double d) { // Parameterized constructor for Box.
width = w;
height = h;
depth = d;
}
double volume( ) {
return width * height * depth;
}
}
class BoxDemo6 {
public static void main(String args[ ]) {
Box mybox1 = new Box(10, 20, 15); // using parameterized constructor
Box mybox2 = new Box(3, 6, 9);
double vol;
vol = mybox1.volume();
System.out.println("Volume is " + vol);
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

**Arrays**

An array is a group of liked-typed variables that are referred to by a common name.
(or) An array is a container object that holds a fixed number of values of a single type.

- Arrays can be of any type and may have one or more dimensions.
- The length of an array is established when the array is created. After creation, its length is fixed.

**One-Dimensional Arrays**

A one-dimensional array is a list of like-typed variables. Usually, creating an array is a two-step process in Java:

1. Creating an array variable of the desired type.
2. Allocating memory for arrays using new operator.

```
For example,
int student_marks[ ]; // creates an array of type int.
month_days = new int [ 10 ]; // Allocates memory
(or)
int student_marks[ ] = new int [ 10 ];
```

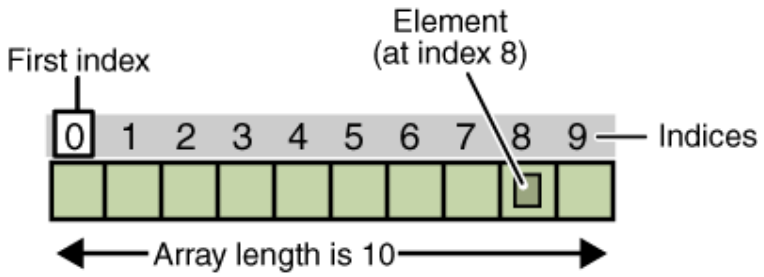



Fig.5: An array of ten elements

Each item in an array is called an element,and each element is accessed by its numerical index.As shown in the above illustration, numbering begins with 0.

The 9th element of the array, for example, would therefore be accessed at index 8.

Declaring an Array Variable.

```
General form:
  type var-name[ ];
   (or)
  type[ ] var-name;

  The following code:
int a[ ];          // declares an array of integers
```

- Like declarations for variables of other types, an array declaration has two components: the array's type and the array's name.
- The square brackets are special symbols indicating that this variable holds an array.
- The size of the array is not part of its type (which is why the brackets are empty).
- An array name can be any valid identifier.

- As with variables of other types, the declaration does not actually create an array — it simply tells the compiler that this variable will refer to an array of the specified type.

```
Similarly, you can declare arrays of other types:
byte b[ ];
char c[ ];
String s[ ];
```

We can also place the square brackets after the array's type name as shown in below statement,

```
float[ ] f;  // f is a variable of type float-array.
```

Sun Microsystems Java code convention recommends the above style of declaring array. Since the brackets identify the array type and should appear with the type designation.

Creating, Initializing, and Accessing an Array
1.One way to create an array is with the new operator :

```
General form :
array-var = new type [ size ] ;
```

The following statements allocates an array with enough memory for 3 integer elements and assigns the array to the "a " variable.

```
  int[ ] a; // Declare array variable.
a = int[3]; //Allocate memory for 3 integers to array a.
```

(or)

Declaration and memory allocation of array in a single line.

```
int a = new int[3];
```

The following lines assign values to each element of the array:

```
a[0] = 100; // initialize first element
a[1] = 200; // initialize second element
a[2] = 300; // initialize third element.
```

Each array element is accessed by its numerical index:

```
System.out.println("Element at index 0 : " + a[0]);
System.out.println("Element at index 1 : " + a[1]);
System.out.println("Element at index 2 : " + a[2]);
```

2.Alternatively, you can use the shortcut syntax to create and initialize an array:

```
int[ ] a = {100, 200, 300, 400, 500 };
```

Here the length of the array is determined by the number of values provided in the initializer list (i.e., between curly braces and ).

**Note:** We can use the built-in length property to determine the size of any array.

The code

```
System.out.println(a.length);
```

Will print the array's size on standard output device.

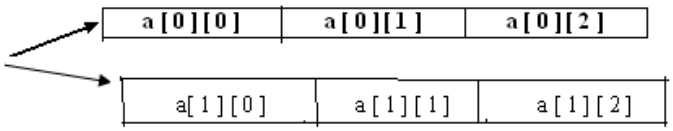
**Multidimensional Arrays:-**

We can also declare an array of arrays (also known as a *multidimensional*array) by using two or more sets of square brackets, such as String [ ][ ] names.

Each element,therefore,must be accessed by a corresponding number of index values.

```
int[ ][ ] a = new int[2][3];
        // Allocates a 2 by 3 array and assigns it to a.
```

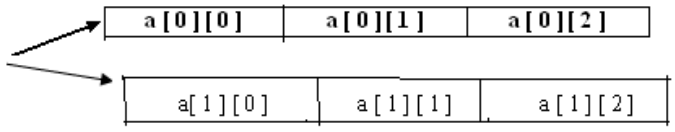
In the Java programming language, a multidimensional array is simply an array whose components are themselves arrays. This is unlike arrays in C or Fortran.




While allocating the memory for multidimensional arrays, we need to only specify the memory for the first (leftmost) dimension. The memory allocation for remaining dimensions can be done separately as shown,

```
int[ ][ ] a = new int[2][ ];
       // First deimension value is compulsory.
a [ 0 ] = new int [ 3 ];
a [ 1 ] = new int [ 3 ];
```

A consequence of this is that the rows are allowed to vary in length.




```
int[ ][ ] a = new int[2][ ]; // First deimension value is compulsory.
a [ 0 ] = new int [ 3 ];
a [ 1 ] = new int [2 ];
```

**Applet**

An applet is a special kind of Java program that a browser enabled with Java technology can download from the internet and run. An applet is typically embedded inside a web page and runs in the context of a browser. An applet must be a subclass of the java.applet.Applet class. The Applet class provides the standard interface between the applet and the browser environment.

Swing provides a special subclass of the Applet class called javax.swing.JApplet. The

JApplet class should be used for all applets that use Swing components to construct their graphical user interfaces (GUIs).

The browser's Java Plug-in software manages the lifecycle of an applet.

**The Applet Class**

Applet provides all necessary support for applet execution, such as starting and stopping. It also provides methods that load and display images, and methods that load and play audio clips. Applet extends the AWT class Panel. In turn, Panel extends Container, which extends Component. These classes provide support for Java's window-based, graphical interface. Thus, Applet provides all of the necessary support for window-based activities.

**An Applet Skeleton**

All but the most trivial applets override a set of methods that provides the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution. Four of these methods, init( ), start( ), stop( ), and destroy( ), apply to all applets and are defined by Applet. Default implementations for all of these methods are provided. Applets do not need to override those methods they do not use. However, only very simple applets will not need to define all of them.

AWT-based applets will also override the paint( ) method, which is defined by the AWT Component class. This method is called when the applet's output must be redisplayed. (Swing-based applets use a different mechanism to accomplish this task.) These five methods can be assembled into the skeleton.

Applet Initialization and Termination

It is important to understand the order in which the various methods shown in the skeleton are called. When an applet begins, the following methods are called, in this sequence:

```
1. init( )
2. start( )
3. paint( )
```

When an applet is terminated, the following sequence of method calls takes place:

1. stop( )
2. destroy( )

**init( )**

The init( ) method is the first method to be called. This is where we should initialize variables. This method is called only once during the run time of our applet.

**start( )**

The start( ) method is called after init( ). It is also called to restart an applet after it has been stopped. Whereas init( ) is called once—the first time an applet is loaded—start( ) is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at start ( ).

**paint( )**

The paint( ) method is called each time our applet's output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored. paint ( ) is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, paint( ) is called. The paint( ) method has one parameter of type Graphics. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

**destroy( )**

The destroy ( ) method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The stop( ) method is always called before destroy( ).

**Overriding update( )**

In some situations, our applet may need to override another method defined by the AWT, called update( ). This method is called when our applet has requested that a portion of its window be redrawn. The default version of update( ) simply calls paint( ). However, you can override the update( ) method so that it performs more subtle repainting.

**Requesting Repainting**

An applet writes to its window only when its update( ) or paint( ) method is called by the AWT. For example, if an applet is displaying a moving banner, what mechanism does the applet use to update the window each time this banner scrolls? One of the fundamental architectural constraints imposed on an applet is that it must quickly return control to

the run-time system. It cannot create a loop inside paint( ) that repeatedly scrolls the banner, for example. This would prevent control from passing back to the AWT. Given this constraint, it may seem that output to your applet's window will be difficult at best. Fortunately, this is not the case. Whenever your applet needs to update the information displayed in its window, it simply calls repaint( ).

The repaint( ) method is defined by the AWT. It causes the AWT run-time system to execute a call to your applet's update( ) method, which, in its default implementation, calls paint( ). Thus, for another part of your applet to output to its window, simply store the output and then call repaint( ). The AWT will then execute a call to paint( ), which can display the stored information. For example, if part of your applet needs to output a string, it can store this string in a String variable and then call repaint( ). Inside paint( ), we will output the string using drawString( ).

The repaint( ) method has four forms. Let's look at each one, in turn. The simplest version of repaint( ) is:

**void repaint( )**

This version causes the entire window to be repainted. The following version specifies a region that will be repainted:

```
void repaint(int left, int top, int width, int height)
```

Where, the coordinates of the upper-left corner of the region are specified by left and top, and the width and height of the region are passed in width and height. These dimensions are specified in pixels. We can save time by specifying a region to repaint. Window updates are costly in terms of time. If you need to update only a small portion of the window, it is more efficient to repaint only that region. Calling repaint( ) is essentially a request that your applet be repainted sometime soon. However, if your system is slow or busy, update( ) might not be called immediately. Multiple requests for repainting that occur within a short time can be collapsed by the AWT in a manner such that update( ) is only called sporadically. This can be a problem in many situations, including animation, in which a consistent update time is necessary. One solution to this problem is to use the following forms of repaint( ):

```
void repaint(long maxDelay)
void repaint(long maxDelay, int x, int y, int width, int height)
```

Where, maxDelay specifies the maximum number of milliseconds that can elapse before update( ) is called. If the time elapses before update( ) can be called, it isn't called. There's no return value or exception thrown.

**System Requirement**

**For Console I/O Applications:**

**Hardware Configuration**

```
Server Configuration :  IBM X3400 Server, Intel Xeon 3.0 GHz CPU,
                        512 KB Cache,2GB DDR RAM
Client Configuration :   Lenovo Dual core 1.60 GHz  / 1 GB RAM
```

**Software Configuration**

```
   Java SE Software :  JDK version 1.6
 Editor         :  Vi editor.
```

**For GUI Applications:**

**Hardware Configuration**

```
Server Configuration :  IBM X3400 Server, Intel Xeon 3.0 GHz CPU,
                                512 KB Cache,2GB DDR RAM
Client Configuration : Lenovo Dual core 1.60 GHz  / 1 GB RAM
```

**Software Configuration**

```
    Java SE Software :  JDK version 1.6
                Editor :  Notepad.
```

# Part II

# Programs

## Program 1

## CONSTRUCTOR AND METHOD OVERLOADING

### Problem Definition

A Program to illustrate the concept of class with constructor and method overloading

### Problem Description

### Method Overloading

The Java programming language supports overloading methods. Method overloading is one of the ways that Java supports polymorphism. Defining two or more methods within the same class that share the same name, but with different parameter declarations are said to be overloaded methods, and the process is referred to as method overloading.

Java can distinguish between methods with different method signatures. This means that methods within a class can have the same name if they have different parameter lists.Suppose that we have a class that can use calligraphy to draw various types of data (strings, integers, and so on) and that contains a method for drawing each data type. It is cumbersome to use a new name for each method—for example, drawString, drawInteger, drawFloat, and so on.

In the Java programming language, we can use the same name for all the drawing methods but pass a different argument list to each method. Thus, the data drawing class might declare four methods named draw, each of which has a different parameter list.

```
public class DataArtist {
...
public void draw(String s) {
...
}
public void draw(int i) {
...
}
public void draw(double f) {
...
}
public void draw(int i, double f) {
...
}
}
```

Overloaded methods are differentiated by the number and the type of the arguments passed into the method. In the code sample, draw(String s) and draw(int i) are distinct and unique methods because they require different argument types.

The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.

### Constructor Overloading

Java allows overloading of constructors to allow initialization of objects in different ways.

Constructor declarations look like method declarations—except that they use the name of the class and have no return type. For example, class Box has one constructor:

```
Box(int w, int h, int d) {
width = w;
height = h;
depth = d;
}
```

To create a new Box object called mybox1:

```
Box mybox1 = new Box (10, 5, 2);
```

A new Box(10, 5, 2) creates the object mybox1 and initializes its width as 10, height as 5 and depth as 2. Although Box class has one constructor, it could have others, including a no-argument constructor:

```
Box( ) { // no-argument constructor
width = 10;
height = 5;
depth = 2;
}
Box mybox2 = new Box( );
```

invokes the no-argument constructor to create a new Box object called mybox2.

Both constructors can be declared in Box because they have different argument lists. Similar to method overloading, the Java platform differentiates overloaded constructors on the basis of the number of arguments in the list and their types.

### Pseudo code

### Method Overloading

Define a class called Box that contains,

1. Three instance variables named width, height and depth of type int.
2. Overload setDim( ) method. Write the following four versions for init( ) method:
   - setDim ( )-Assign the width to 1, height to 1 and depth to 1.
   - setDim (int)–Assign the width with received value and height and depth to 1.

- setDim (int, int)–Assign the width, height with received values and depth to 1.
- setDim (int, int, int)–Assign the width, height and depth with received values.

3. Define a method volume( ) that returns the volume of Box object.

**/* Program to demonstrate Method Overloading. */**

```
class Box {
// Declare instance variables here (step-1 of pseudo code).
//Implement the four versions of setDim ( ) method here
                      (step-2 of pseudo code).
// Implement the volume( ) method here (step-3 of pseudo code).
}


class MODemo {

public static void main(String[ ] args)  {

Box b1 = new Box( );

b1. setDim ( );
System.out.println("Volume of box b1 is "+b1.volume( ));

b1. setDim (8);
System.out.println("Volume of box b1 is "+b1.volume( ));

b1. setDim (8,6);
System.out.println("Volume of box b1 is "+b1.volume( ));

b1. setDim (8,6,4);
System.out.println("Volume of box b1 is "+b1.volume( ));
  }
     }
```

**Constructor Overloading**

Create a class called Box that contains,

1. Three instance variables named width, height and depth.
2. Overload constructor for Box class with the following five versions:
   - Box( )-No-argument constructor that initializes the width to 1, height to 1 and depth to 1. Box(int)–Parameterized Constructor that initializes the width with received value. Sets height and depth to 1. Box(int, int)–Parameterized Constructor that is used to initialize the width and height with received values. Set depth to 1. Box(int, int, int)–Parameterized Constructor that initializes the width, height and depth with received values. Box(Box b)– Constructor that initializes the width, height and depth of current Box object with width, breadth and height of already existing Box object b respectively. This constructor is also called Copy constructor.
3. Define a method volume ( ) that returns the volume of Box object.

**/* Program to demonstrate Constructor Overloading. */**

```
class Box {

// Declare instance variables here (step-1 of pseudo code).
// Implement all the overloaded versions of constructors here
                  (step-2 of pseudo code).
// Implement volume( ) method here  (step-3 of pseudo code)..
}


class CODemo {

public static void main(String[ ] args)  {

Box b1 = new Box( );
System.out.println("Volume of box b1 is "+b1.volume( ));

Box b2 = new Box(8);
System.out.println("Volume of box b2 is "+b2.volume( ));

Box b3 = new Box(8,6);
System.out.println("Volume of box b3 is "+b3.volume( ));

Box b4 = new Box(8,6,4);
System.out.println("Volume of box b4 is "+b4.volume( ));
```

```
Box b5 = new Box(b4);
System.out.println("Volume of box b5 is "+b5.volume( ));
}
}
```

### Problem Validation

### Method Overloading

```
Input:

a. create an object b1 of Box class.
b. Assign b1's width=1, height=1 and depth = 1.
c. Assign b1's width=8, height=1 and depth = 1.
d. Assign b1's width=8, height=6 and depth = 1.
e. Assign b1's width=8, height=6 and depth = 4.

Output:

Volume of box b1 is 100
Volume of box b1 is 8
Volume of box b1 is 48
Volume of box b1 is 192
```

### Constructor Overloading

```
Input:
a.Create an object b1 of Box class initializing its
   width = 1, height = 1 and depth = 1 $($use no-argument constructor$)$.
b. Create an object b2 of Box class initializing its
   width = 8, height = 1 and depth = 1.
c. Create an object b3 of Box class initializing its
   width = 8, height = 6 and depth = 1.
d. Create an object b4 of Box class initializing its
   width = 8, height = 1 and depth = 4.
e. Create an object b5 of Box class which is replica of b4 object
   $($using copy constructor$)$.

Output:
Volume of box b1 is 1
Volume of box b2 is 8
Volume of box b3 is 48
Volume of box b4 is 192
Volume of box b5 is 192
```

**Program 2**

## INHERITANCE AND DYNAMIC POLYMORPHISM

### Problem Definition

A program to illustrate the concept of inheritance and dynamic polymorphism.

### Problem Description

### Inheritance

Central to Java and other object-oriented (OO) languages is the concept of inheritance, which allows code defined in one class to be reused in other classes. Java, inheritance is achieved with the keyword "extends".

A class that is derived from another class is called a subclass (also a derived class, or a child class). The class from which the subclass is derived is called a superclass (also a base class or a parent class).

Fig. 6: Diagrammatic Representation of Inheritance

A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

### Polymorphism

The dictionary definition of polymorphism refers to a principle in biology in which an organism or species can have many different forms or stages. This principle can also be applied to object-oriented programming and languages like the Java language. Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.

Java implements Dynamic (run-time) polymorphism through 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.

When a method in a subclass has the same name and type signature as a method in its super class, then the method in the subclass is said to override the method in the super class.

An important principle in java is that it allows a super class reference variable to refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time.

When an overridden method is called through a super class reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.

In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.

**Pseudo code**

**Inheritance**

1. 1. Create a class called A that contains,
    - Two instance variables named i and j of type int.
    - Define a method showij( ) that display the values of i and j variable.
2. Create another class called B extending class A that contains,
    - Declare an instance variable say k of type int.
    - Define a method showk ( ) that display the value of k.
    - Define another method called sum( ) that display the sum of values of i, j and k.

**/* Java program to illustrate the concept of inheritance. */**

```
class A {
// Implement class A based on step-1 of pseudo code.
// methods and variables must not be declared as private.
}
class B extends A {
// Similarly, Implement class B based on step-2 of pseudo code.
// methods and variables must not be declared as private.
}
  class SimpleInheritance {
   public static void main(String args[ ] )  {
    A superOb = new A( );
    B subOb = new B( );
    superOb.i = 5;
    superOb.j = 6;
    System.out.println("Contents of superOb: ");
    superOb.showij( );
    System.out.println( );
    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( );
   }
}
```

**Polymorphism**

1. 1. Create a class called Figure that contains,
   - Two instance variables named dim1 and dim2 of type double.
   - A parameterized constructor to initialize both dim1 and dim2 instance variables.
   - Define a method area( ) that returns zero since the area is undefined for Figure.
2. Create another class called Rectangle extending class Figure that contains,
   - A parameterized constructor which in turn calls the constructor of super class (class Figure) to initialize dim1 and dim2.
   - Overrides the area( ) method to calculate the area value of rectangle ( dim1 $*$ dim2 ).
3. Create another class called Triangle extending class Figure that contains,
   - a.A parameterized constructor which in turn calls the constructor of super class (class Figure) to initialize dim1 and dim2.
   - b.Overrides the area( ) method to calculate the area value of triangle(1/2 dim1*dim2).

**/* Java program to demonstrate dynamic polymorphism. */**

```
class Figure {
// Implement class Figure based on step-1 of pseudo code.
}
class Rectangle extends Figure {
// Implement class Rectangle based on step-2 of pseudo code.
}
class Triangle extends Figure {
// Implement class Triangle based on step-3 of pseudo code.
}
class FindAreas {
public static void main(String args[ ]) {
Figure f = new Figure(10, 10);
Rectangle r = new Rectangle(10, 5);
Triangle t = new Triangle(10, 8);
Figure figref;
figref = r;
System.out.println("Area is " + figref.area());
figref = t;
System.out.println("Area is " + figref.area());
```

```
figref = f;
System.out.println("Area is " + figref.area());
}
}
```

**Problem Validation**

**Inheritance**

**Input:**

```
a.Create two objects say superOb and subOb for class A and class B
   respectively.
b.Set i and j value of superOb to 5 and 6.
c.Set i, j and k value of subOb to 7, 8 and 9.
```

**Output:**

```
Contents of superOb:
i and j: 5 6
Contents of subOb:
i and j: 7 8
k: 9
Sum of i, j and k in subOb:
i+j+k: 24
```

**Polymorphism**

**Input:**

```
a.Create object of Figure initializing dim1 and dim2 to 10 and 10
   respectively.
b.Create object of Rectangle initializing dim1 and dim2 to 10 and 5
   respectively.
c.Create object of Figure initializing dim1 and dim2 to 10 and 8
   respectively.
```

**Output:**

```
Inside Area for Rectangle.
Area is 50.0
Inside Area for Triangle.
Area is 40.0
Area for Figure is undefined.
Area is 0.0
```

**Program 3**

## ABSTRACT CLASS

### Problem Definition

A program to illustrate the usage of abstract class.

### Problem Description

### Abstract Class

An abstract class is a class that is declared abstract. It may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon). For example,

```
abstract void moveTo(double x, double y);
```

If a class includes abstract methods, the class itself must be declared abstract, as in:

```
abstract class GraphicObject {
   // declare fields
   // declare non-abstract methods
   abstract void draw( );
}
```

When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, the subclass must also be declared abstract.

### Pseudo code

1. Create a class called Figure that contains,
   - Two instance variables named dim1 and dim2 of type double.
   - A parameterized constructor to initialize both dim1 and dim2 instance variables.
   - Declare a method area( ) without implementation. Since Figure is too general and its area is undefined.
2. Create another class called Rectangle extending class Figure that contains,
   - A parameterized constructor which calls constructor of Figure class.
   - Overrides the area( ) method to calculate the area value of rectangle (dim1*dim2).

3. Create another class called Triangle extending class Figure that contains,

   - A parameterized constructor which calls constructor of Figure class.
   - Overrides the area( ) method to calculate the area value of triangle (1/2 dim1*dim2).

**/* Program to Demonstrate abstract class concept. */**

```
abstract class Figure {

  // Declare instance variables here (step-1.a of pseudo code).

// Write a parameterized constructor to initialize dim1 and dim2 instance
// variables (step-1.b of pseudo code).
abstract double area( );  /* area is now an abstract method */
}

class Rectangle extends Figure {

// Implement class Rectangle based on step-2 of pseudo code.
}
class Triangle extends Figure {

// Implement class Triangle based on step-3 of pseudo code.
}

class AbstractAreas {

public static void main(String args[ ]) {

  // Figure f = new Figure(10, 10); /* illegal now */
Rectangle r = new Rectangle(10, 5);
Triangle t = new Triangle(10, 8);
Figure figref; /* this is OK, no object is created. */

figref = r;
System.out.println("Area is " + figref.area());

figref = t;
System.out.println("Area is " + figref.area());
}
}
```

**Problem Validation**

```
Input:
a.Create objects for concrete classes i.e., Rectangle and Triangle.
b.Initialize rectangle's dim1 and dim2 to 10 and 5 respectively.
c.Initialize Triangle's dim1 and dim2 to 10 and 8 respectively.

Output:

Inside Area for Rectangle.
Area is 50.0
Inside Area for Triangle.
Area is 40.0
```

**Program 4**

## MULTI THREADING

### Problem Definition

A program to illustrate multithreading.

### Problem Description

### Multi threading

Java provides built-in support for *multithreaded programming*. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

There are two distinct types of multitasking:

- process-based and
- thread-based.

*process-based* **multitasking** is the feature that allows your computer to run two or more programs concurrently.

In a *thread-based* **multitasking** environment, the thread is the smallest unit of dispatchable code.

This means that a single program can perform two or more tasks simultaneously.

| Process | Threads |
|---|---|
| Processes are heavyweight tasks that require their own separate address spaces. | Threads are lightweight. They share the same address space and cooperatively share the same heavyweight process. |
| Inter-process communication is expensive and limited. | Inter-thread communication is inexpensive. |
| Context switching from one process to another is also costly. | Context switching from one thread to the next is low cost. |

Figure 1: T
able 2: Process vs Thread

Thus, Multitasking threads require less overhead than multitasking processes. Multithreading enables you to write very efficient programs that make maximum use of the

CPU, because idle time can be kept to a minimum. This is especially important for the interactive, networked environment in which Java operates, because idle time is common.

### Thread States

A thread can be in one of the following states:

| State | Description |
|---|---|
| NEW | A thread that has not yet started is in this state. |
| RUNNABLE | A thread executing in the Java virtual machine is in this state. |
| BLOCKED | A thread that is blocked waiting for a monitor lock is in this state. |
| WAITING | A thread that is waiting indefinitely for another thread to perform a particular action is in this state. |
| TIMED_WAITING | A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state. |
| TERMINATED | A thread that has exited is in this state. |

**Table 3: States of a Thread in Java**

### Java's multithreading system

Java's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**. Java defines two ways to create user-defined threads.

- Implementing the Runnable interface.
- Extending the Thread class.

### Implementing Runnable

To implement Runnable, a class need only implement a single method called run( ), which is declared like this:

```
public void run( )
```

Inside **run( )**, we must define the code that constitutes the new thread. It is important to understand that **run( )** can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that **run( )** establishes the entry point for another, concurrent thread of execution within our program. This thread will end when **run( )** returns.

Procedure to create a user-defined thread using Runnable interface:

1. Write a class that implements Runnable interface.
2. Create a Thread class object within this class using the following constructor, Thread ( Runnable threadOb, String threadName ) Here threadOb is an instance of a class whose run method will be executed. The name of the new thread is specified by threadName.

3. Then call the start( ) method on thread object to begin the execution of newly created thread.

**Extending Thread**

The second way to create a user-defined thread is by extending java.lang.Thread class.

Procedure to create a user-defined thread using Thread class:

1. Create a new class that extends Thread.
2. The extending class must override the run( ) method, which is the entry point for the new thread.
3. Then call start( ) to begin the execution of the new thread.

**Pseudo code**

**Implementing Runnable interface**

1. Create a class NewThread implementing Runnable interface.
2. Create a Thread object.
3. Start the thread object calling start( ) method on it.
4. Implement run( ) method to display numbers from 5 to 1 with a delay of 0.5 second. [Hint : Delay can be obtained using sleep( ) method of java.lang.Thread class.]

**/* Program to create a user-defined thread implementing Runnable interface. */**

```
class NewThread implements Runnable {
 Thread t;
 NewThread( ) {
    //Create  new Thread object (step-2 of pseudo code).
    // Start the thread (step-3 of pseudo code)..
  }
  public void run( ) { /* Entry point for the user-defined thread. */
// Implement code here to display numbers from 5 to 1 with
// delay of 0.5 seconds (step-4 of pseudo code).
}
}
class UserThreadDemo {
  public static void main(String args[ ]) {
new NewThread( );
    try {
      for(int i = 5; i > 0; i--) {
```

```
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {

      System.out.println("Main thread interrupted.");
    }
    System.out.println("Main thread exiting.");
  }
}
```

**Extending Thread class**

1. Create a class NewThread Extending Thread class.
2. Start the thread object.
3. Implement run( ) method to display numbers from 5 to 1 with a delay of 0.5 second.

**Extending Thread**
**/* Creates a user-defined thread by extending Thread class. */**

```
class NewThread extends Thread {
 NewThread( ) {
    super("Demo Thread");
    // Start the thread (step-2 of pseudo code).
   }
 public void run( ) {   /* Entry point for the user-defined thread.*/
// display numbers from 5 to 1 with delay of 0.5 seconds -
// - (step-3 of pseudo code).
 }
}
class UserThreadDemo {
  public static void main(String args[ ] ) {
    new NewThread( ); /* creates a new thread.*/
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
       System.out.println("Main thread interrupted.");
    }
    System.out.println("Main thread exiting.");
  }
}
```

**Problem Validation**

**Implementing Runnable Interface**

```
Input:
N/A
Output:
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Child Thread: 3
Main Thread: 4
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
 Extending Thread Class


Input:
N/A
Output:
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Child Thread: 1
Main Thread: 3
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

Note: Output may differ since the thread scheduling by JVM is unpredictable.

**Program 5**

## THREAD SYNCHRONIZATION

### Problem Definition

A program to illustrate thread synchronization.

### Problem Description

### Synchronization

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called *synchronization.* Key to synchronization is the concept of the monitor (also called a *semaphore*). A *monitor* is an object that is used as a mutually exclusive lock, or *mutex.* Only one thread can *own* a monitor at a given time. When a thread acquires a lock, it is said to have *entered* the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread *exits* the monitor. These other threads are said to be *waiting* for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires.

The Java programming language provides two basic synchronization idioms:

- synchronized methods
- synchronized statements

### synchronized methods

To make a method *synchronized*, simply add the synchronized keyword to its declaration. While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait. To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment ( ) {
        c++;
    }
    public synchronized void decrement( ) {
        c--;
    }
    public synchronized int value( ) {
        return c;
    }
}
```

Note that constructors cannot be synchronized — using the synchronized keyword with a constructor is a syntax error. Synchronizing constructors doesn't make sense, because only the thread that creates an object should have access to it while it is being constructed.

**synchronized statements**

Another way to create synchronized code is with synchronized statements. Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock: General Form,

```
synchronized(object) {
// statements to be synchronized
}
For Example,
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

Here, object is a reference to the object being synchronized. A synchronized block ensures that a call to a method that is a member of object occurs only after the current thread has successfully entered object's monitor.

Synchronized statements are useful when third party classes are used and we do not have access to its source code. Synchronized statements are also useful for improving concurrency with fine-grained synchronization. Instead of making whole method as synchronized, only the critical section can be synchronized using synchronized statement.

**Interthread Communication**

The use of the implicit monitors in Java objects is powerful, but we can achieve a more subtle level of control through interprocess communication. Multithreading replaces event loop programming by dividing your tasks into discrete, logical units. Threads also provide a secondary benefit: they do away with polling. Polling is usually implemented by a loop that is used to check some condition repeatedly. Once the condition is true, appropriate action is taken. This wastes CPU time.

For example, consider the classic queuing problem, where one thread is producing some data and another is consuming it. To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data. In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce. Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish, and so on.

To avoid polling, Java includes an elegant interprocess communication mechanism via the wait( ), notify( ), and notifyAll( ) methods. These methods are implemented as final methods in Object, so all classes have them. All three methods can be called only within a synchronized context.

- wait( ) tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ).
- notify( ) wakes up a thread that called wait( ) on the same object.
- notifyAll( ) wakes up all the threads that called wait( ) on the same object. One of the threads will be granted access.

These methods are declared within Object class, as:

final void wait( ) throws InterruptedException

final void notify( )

final void notifyAll( )

**Pseudocode**

1. Create a class called Q that contains,
   - An integer variable say "n" (n is considered a queue that can store only one value)
   - A boolean variable say "available". When available is true it indicates that n contains a value, otherwise n doesn't contain a value.
   - A synchronized get ( ) method. This method will be invoked by Consumer thread to consume value stored in "n". Value must be given to Consumer thread only if available is true. Otherwise it must wait until the value of available is false.
   - A synchronized put ( ) method. This method will be invoked by Producer thread to produce/store a value in "n". New value from Producer thread must be accepted only if available is false. Otherwise it must wait until the available becomes false.
2. Create a Producer thread. Use this producer thread to produce infinite values
3. Create a Consumer thread. Use this consumer thread to consume the values.

**// Implementation of producer and consumer problem.**

```
class Q {
  // Declaration stmt goes here (step-1.a and step1.b of pseudo code).
  // Implement synchronized get( ) method here (step-1.c of pseudo code).
  // Implement synchronized put( ) method here (step-1.d of pseudo code).
     }
class Producer implements Runnable {
  Q q;
  Producer(Q q) {
    this.q = q;
    new Thread(this, "Producer").start( );
  }
  public void run( ) {
    // invoke put( ) method infinite number of times to produce values.
   // (step-2 of pseudo code).
  }
}
class Consumer implements Runnable {
  Q q;
  Consumer(Q q) {
    this.q = q;
    new Thread(this, "Consumer").start( );
  }
  public void run() {
 // Consume values invoking get( ) method infinitely
     (step-3 of pseudo code)..
  }
}
class PCFixed {
  public static void main(String args[ ]) {
    Q q = new Q( );
    new Producer(q);
    new Consumer(q);
    System.out.println("Press Control-C to stop.");
  }
}
```

### Problem Validation

```
Input:
N/A
Output:
Put: 1
Got: 1
Put: 2
Got: 2
```

```
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
   .
   .
   .
```

## Program 6

## STRING TOKENIZER

### Problem Definition

A program using StringTokenizer.

### Problem Description

### StringTokenizer

The StringTokenizer is a predefined class available in java.util package. The string tokenizer class allows an application to break a string into tokens. The tokenization method is much simpler than the one used by the StreamTokenizer class. The StringTokenizer methods do not distinguish among identifiers, numbers, and quoted strings, nor do they recognize and skip comments.

The set of delimiters (the characters that separate tokens) may be specified either at creation time or on a per-token basis. An instance of StringTokenizer behaves in one of two ways, depending on whether it was created with the returnDelims flag having the value true or false:

- If the flag is false, delimiter characters serve to separate tokens. A token is a maximal sequence of consecutive characters that are not delimiters.
- If the flag is true, delimiter characters are themselves considered to be tokens. A token is thus either one delimiter character, or a maximal sequence of consecutive characters that are not delimiters.

The StringTokenizer contains the following constructors:

| S. No. | Constructors | Description |
|---|---|---|
| 1. | **StringTokenizer**(String str) | Constructs a string tokenizer for the specified string. |
| 2. | **StringTokenizer**(String str, String delim) | Constructs a string tokenizer for the specified string with *delim* as a string that specifies the delimiters. |
| 3. | **StringTokenizer**(String str, String delim, boolean returnDelims ) | Constructs a string tokenizer for the specified string with *delim* as a string that specifies the delimiters. If returnDelims is **true**, then the delimiters are also returned as tokens when the string is parsed. Otherwise, the delimiters are not returned. Delimiters are not returned as tokens by the first two forms. |

**Table 4: Constructors of StringTokenizer**

A StringTokenizer object internally maintains a current position within the string to be tokenized. Some operations advance this current position past the characters processed.

| S. No. | Method | Description |
|---|---|---|
| 1. | int countTokens( ) | Calculates the number of times that this tokenizer's nextToken method can be called before it generates an exception. |
| 2. | boolean hasMoreElements( ) | Returns the same value as the hasMoreTokens method. |
| 3. | boolean hasMoreTokens( ) | Tests if there are more tokens available from this tokenizer's string. |
| 4. | Object nextElement( ) | Returns the same value as the nextToken method, except that its declared return value is Object rather than String. |
| 5. | String nextToken( ) | Returns the next token from this string tokenizer. |
| 6. | String nextToken (String delim) | Returns the next token in this string tokenizer's string. |

**Table 5: Methods of StringTokenizer**

A token is returned by taking a substring of the string that was used to create the StringTokenizer object. Methods of StringTokenizer class,

**Pseudo code**

1. Create a string object storing some content.
2. Create a StringTokenizer for the string to generate tokens based on space/tab-space/ new-line characters.
3. Print all the tokens generated.
4. Create a StringTokenizer that generate tokens based on delimiters semicolon(;) and comma (,).
5. Print all the tokens generated.
6. Create a StringTokenizer that generate tokens based on delimiters colon (:) and comma (,) considering delimiters as tokens.
7. Print all the tokens generated.

**/* Java program using StringTokenizer.*/**

```
import java.util.*;
class StringTokenizerDemo {
public static void main(String[ ] args) {
String str1 = "Branch:CSE, Course:BE, College:MJCET.";
// 1. Generate and print tokens based on space
// 2. Generate and print tokens based on : and ,
// 3. Generate and print tokens based on colon (:) and comma (,)
//  considering : and , as tokens.
}
}
```

**Problem Validation**

```
Input:
Input String is "Branch:CSE, Course:BE, College:MJCET".
Output:
String tokens using st1
Branch:CSE,
Course:BE,
College:MJCET.

String tokens using st2
Branch
CSE
 Course
BE
 College
MJCET.

String tokens using st3
Branch
:
CSE
,
 Course
:
BE
,
 College
:
MJCET.
```

## Program 7

## LINKED LIST

### Problem Definition

A program using Linked list class.

### Problem Description

### LinkedList

The LinkedList class is defined in java.util package. The LinkedList class is a part of collection framework.


Fig. 7: Hierarchy of List interface

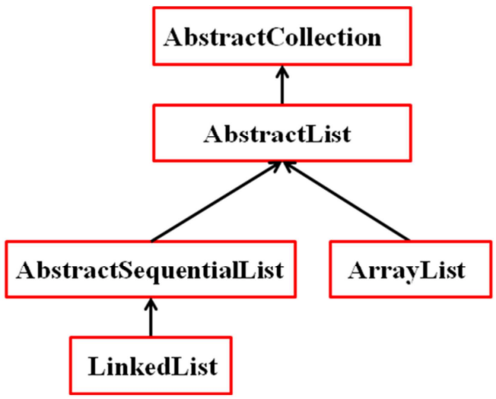



Fig. 8: Hierarchy of List Classes

It is the Linked list implementation of the List interface. Implements all optional list operations, and permits all elements (including null). In addition to implementing the List interface, the LinkedList class provides uniformly named methods to get, remove and insert an element at the beginning and end of the list. These operations allow linked lists to be used as a stack, queue, or double-ended queue.

The class implements the Deque interface, providing first-in-first-out queue operations for add, poll, along with other stack and deque operations.
LinkedList is a generic class that has this declaration:

```
class LinkedList<E>
```

Where, E specifies the type of objects that the list will hold.

LinkedList has the two constructors:

| S. No. | Constructor | Description |
|---|---|---|
| 1. | LinkedList( ) | Constructs an empty list. |
| 2. | LinkedList(Collection <? extends E> c) | Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator. |

**Table 6: Constructor of LinkedList**

Table 6: Constructor of LinkedList

Because LinkedList implements the Deque interface, we have access to the methods defined by Deque. For example, to add elements to the start of a list we can use addFirst( ) or offerFirst( ). To add elements to the end of the list, use addLast( ) or offerLast( ). To obtain the first element, we can use getFirst( ) or peekFirst( ). To obtain the last element, use getLast( ) or peekLast( ). To remove the first element, use removeFirst( ) or pollFirst( ). To remove the last element, use removeLast( ) or pollLast( ).

All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

Few methods of LinkedList:

| S. No. | Method | Description |
|---|---|---|
| 1. | boolean add(E e) | Appends the specified element to the end of this list. |
| 2. | void addFirst(E e) | Inserts the specified element at the beginning of this list. |
| 3. | void addLast(E e) | Appends the specified element to the end of this list. |
| 4. | void add(int index, E element) | Inserts the specified element at the specified position in this list. |
| 5. | E removeFirst( ) | Removes and returns the first element from this list. |
| 6. | E removeLast( ) | Removes and returns the last element from this list. |
| 7. | E remove(int index) | Removes the element at the specified position in this list. |
| 8. | E get(int index) | Returns the element at the specified position in this list. |
| 9. | E set(int index, E element) | Replaces the element at the specified position in this list with the specified element. |

**Table 7: Methods of LinkedList**

**Pseudo code**

1. Create a LinkedList that can store string objects.
2. Add the following string objects to the list, “F”, “G”, “D”, “E”, “C”.
3. Print the list contents.
4. Add string objects “A” and “Z” at first and last position of the list.
5. Add string object “B” at index 1 in the list.
6. Print the list.
7. Remove first and last element from the list.
8. Remove element at index 2.
9. Print the list.
10. Get the element at index 3 and print.
11. Update the element at index 3 to “X”.

```
/* Demonstrate LinkedList. */
import java.util.*;
class LinkedListDemo {
    public static void main(String args[ ]) {
// Implement pseudo code steps here...
    }
}
```

**Problem Validation**

**Input:**
N/A **Output:**

```
Initial List Contents are : [F, G, D, E, C]
After insertion, List contents are : [A, B, F, G, D, E, C, Z]
After deletion, List contents are : [B, F, D, E, C]
Element at index 3 is : E
After updation, List contents are : [B, F, D, X, C]
```

## Program 8

## TREESET

### Problem Definition

A program using TreeSet class.

### Problem Description

### TreeSet

The Set interface defines a set. It extends Collection and declares the behavior of a collection that does not allow duplicate elements. Therefore, the add( ) method returns false if an attempt is made to add duplicate elements to a set.

TreeSet extends AbstractSet and implements the NavigableSet interface. It creates a collection that uses a tree for storage. The elements are ordered using their natural ordering, or by a Comparator provided at set creation time, depending on which constructor is used.

Access and retrieval times are quite fast, which makes TreeSet an excellent choice when storing large amounts of sorted information that must be found quickly. This implementation provides guaranteed log(n) time cost for the basic operations (add, remove and contains).

TreeSet is a generic class that has this declaration:

class TreeSet¡E¿
where, E specifies the type of objects that the set will hold.

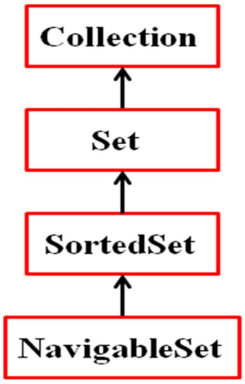



Fig. 9: Hierarchy of Set interface

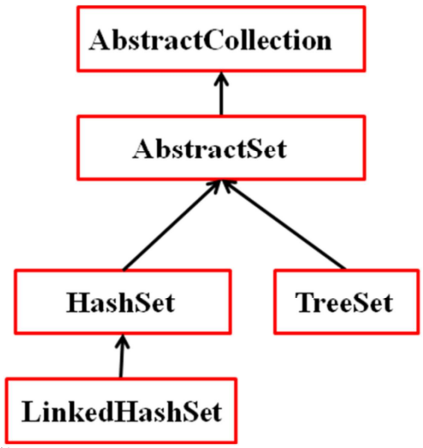



Fig. 10: Hierarchy of Set Classes

TreeSet has the following constructors:

| S. No | Constructor | Description |
|---|---|---|
| 1. | TreeSet( ) | Constructs a new, empty tree set, sorted according to the natural ordering of its elements. |
| 2. | TreeSet(Collection<? extends E> *c*) | Constructs a new tree set containing the elements in the specified collection, sorted according to the *natural ordering* of its elements. |
| 3. | TreeSet(Comparator<? super E> *comp*) | Constructs a new, empty tree set, sorted according to the specified comparator. |
| 4. | TreeSet(SortedSet<E> *ss*) | Constructs a new tree set containing the same elements and using the same ordering as the specified sorted set. |

**Table 8: Constructor of TreeSet**

### Pseudo code

1. Create a TreeSet to store string objects.
2. Add the following string objects to tree set: “D”, “B”, “E”, “A”, “C”, “F”
3. Print the elements of tree set.
4. Print the elements that are before element “D”.
5. Print the elements that are after element “D”.

6. Print the elements between element “B” and “E”.

**/* Java program using TreeSet class.*/**

```
import java.util.*;

class TreeSetDemo {

public static void main(String[ ] args) {

// Implement Pseudo code steps here...
}
}
```

**Problem Validation**

**Input:**

```
N/A
```

**Output:**

```
[A, B, C, D, E, F]
[A, B, C]
[D, E, F]
[B, C, D]
```

## Program 9

## HASHSET AND ITERATOR

### Problem Definition

A program using HashSet and Iterator classes.

### Problem Description

### HashSet

**HashSet** extends AbstractSet and implements the Set interface. It creates a collection that uses a hash table for storage.

**HashSet** is a generic class that has this declaration:

class HashSet < E > Where,E specifies the type of objects that the set will hold.

Hash table stores information by using a mechanism called hashing. In *hashing*, the informational content of a key is used to determine a unique value, called its hash code. The *hash code* is then used as the index at which the data associated with the key is stored. The transformation of the key into its hash code is performed automatically—you never see the hash code itself. Also, your code can't directly index the hash table. The advantage of hashing is that it allows the execution time of **add( ), contains( ), remove( ), and size( )** to remain constant even for large sets.

The following constructors are defined:

| S. No | Constructor | Description |
|---|---|---|
| 1. | HashSet( ) | Constructs a default hash set. Default initial capacity (16) and load factor (0.75). |
| 2. | HashSet(Collection<? extends E> *c*) | Initializes the hash set by using the elements of *c*. |
| 3. | HashSet(int *capacity*) | Initializes the capacity of the hash set to *capacity* (The default capacity is 16). |
| 4. | HashSet(int *capacity*, float *fillRatio*) | Initializes both the capacity and the fill ratio (also called *load capacity*) of the hash set from its arguments. The fill ratio must be between 0.0 and 1.0, and it determines how full the HashSet can be before it is resized upward. Specifically, when the number of elements is greater than the capacity of the hash set multiplied by its fill ratio, the hash set is expanded. For constructors that do not take a fill ratio, 0.75 is used. |

**Table 9: Constructor of HashSet**

It is important to note that HashSet does not guarantee the order of its elements, because the process of hashing doesn't usually lend itself to the creation of sorted sets.

**Iterators** Often, we want to cycle through the elements in a collection. For example, we might want to display each element. One way to do this is to employ an iterator, which is an object that implements either the **Iterators** or the **ListIterator** interface. Iterator enables you to cycle through a collection, obtaining or removing elements. **ListIterator**

extends Iterator to allow bidirectional traversal of a list, and the modification of elements. Iterator and **ListIterator** are generic interfaces which are declared as:

```
interface Iterator <E>
```

interface ListIterator < E >

Where, E specifies the type of objects being iterated.

The**Iterator** interface declares the following methods

| S. No | Method | Description |
|---|---|---|
| 1. | boolean hasNext( ) | Returns true if the iteration has more elements. |
| 2. | E next( ) | Returns the next element in the iteration. |
| 3. | E remove( ) | Removes from the underlying collection the last element returned by the iterator (optional operation). |

**Table 10: Methods of Iterator**

Before we can access a collection through an iterator, we must obtain one. Each of the collection classes provides an iterator( ) method that returns an iterator to the start of the collection. By using this iterator object, we can access each element in the collection, one element at a time. In general, to use an iterator to cycle through the contents of a collection, follow these steps:

1. Obtain an iterator to the start of the collection by calling the collection's iterator( ) method.
2. Set up a loop that makes a call to hasNext( ). Have the loop iterate as long as hasNext( ) returns true.
3. Within the loop, obtain each element by calling next( ).

For collections that implement **List**, we can also obtain an iterator by calling **listIterator()**.

**Pseudocode**

1. Create a HashSet of string objects.
2. Add objects "D", "B", "E", "A", "C", "F" to hash set.
3. Print the hash set.
4. Get the Iterator object on hash set.
5. Print the hash set objects using iterator.

| S. No | Method | Description |
|---|---|---|
| 1. | void add(E e) | Inserts the specified element into the list (optional operation). |
| 2. | boolean hasNext( ) | Returns true if this list iterator has more elements when traversing the list in the forward direction. |
| 3. | boolean hasPrevious( ) | Returns true if this list iterator has more elements when traversing the list in the reverse direction. |
| 4. | E next() | Returns the next element in the list. |
| 5. | int nextIndex( ) | Returns the index of the element that would be returned by a subsequent call to next. |
| 6. | E previous( ) | Returns the previous element in the list. |
| 7. | int previousIndex( ) | Returns the index of the element that would be returned by a subsequent call to previous. |
| 8. | void remove( ) | Removes from the list the last element that was returned by next or previous (optional operation). |
| 9. | void set(E e) | Replaces the last element returned by next or previous with the specified element (optional operation). |

**Table 11: Methods Defined by ListIterator**

6. Create an ArrayList adding all elements exist in hash set.
7. Get the ListIterator object on array list.
8. Print the elements of array list in forward direction.
9. Print the elements of array list in backward direction.

**/* java program using HashSet and Iterator classes.*/**

```
import java.util.*;
class HashSetAndIterator {
public static void main(String[ ] args) {
// Create HashSet (step-1 of pseudo code).
// write code here for adding objects ( step-2 of pseudo code).
// print hash set (step-3 of pseudo code).
System.out.println("Iterating through Elements of Hash Set");
// Get the iterator on hash set and print each object through iterator
// (step-4 and step-5 of pseudo code).
System.out.println( );
// Create ArrayList here (step-6 of pseudo code).
// Get list iterator on array list and print elements in both forward and
// backward direction (step-7 to step-9 of pseudo code).
}
}
```

**Problem Validation**

```
Input:
N/A
Output:

[A, B, C, D, E, F]
Iterating through Elements of Hash Set
A
B
C
D
E
F

List contents in forward direction : A B C D E F
List contents in backward direction : F E D C B A
```

## Program 10

## MAP CLASSES

### Problem Definition

A program using map classes.

### Problem Description

### Map

A map is an object that stores associations between keys and values, or key/value pairs. Given a key, we can find its value. Both keys and values are objects. The keys must be unique, but the values may be duplicated. Some maps can accept a null key and null values, others cannot.

The Map Interfaces

The following interfaces support maps:

| Interface | Description |
|---|---|
| Map | Maps unique keys to values. |
| Map.Entry | Describes an element (a key/value pair) in a map. This is an inner class of Map. |
| NavigableMap | Extends SortedMap to handle the retrieval of entries based on closest-match searches.(Added by Java SE 6.) |
| SortedMap | Extends Map so that the keys maintained in ascending order. |

Table 12: Map Interfaces

### The Map Interface

The **Map** interface maps unique keys to values. A key is an object that we use to retrieve a value at a later date. Given a key and a value, we can store the value in a **Map** object. After the value is stored, we can retrieve it by using its key. **Map** is generic and is declared as:

```
interface Map<K, V>
```

Where, **K** specifies the type of keys, and **V** specifies the type of values.

Maps revolve around two basic operations: **get( )** and **put( )**. To put a value into a map,use put( ), specifying the key and the value. To obtain a value, call **get( )**, passing the key as an argument. The value is returned.

Although Maps are not a part of the Collections Framework and do not implement the Collection interface. However, we can obtain a collection-view of a map. To do this, we can use the **entrySet( )** method. It returns a Set that contains the elements in the map. To obtain a collection-view of the keys use **keySet( )**. To get a collection-view of the values, use **values()**. Collection-views are the means by which maps are integrated into the larger Collections Framework.

| S.No | Methods | Description |
|---|---|---|
| 1. | V get(Object k) | Returns the value associated with the key k. Returns null if the key is not found. |
| 2. | V put(K k, V v) | Puts an entry in the invoking map, overwriting any previous value associated with the key. The key and value are k and v, respectively. Returns null if the key did not already exist. Otherwise, the previous value linked to the key is returned. |
| 3. | Set<Map.Entry<K, V>> entrySet( ) | Returns a Set that contains the entries in the map. The set contains objects of type Map.Entry. Thus, this method provides a set-view of the invoking map. |
| 4. | boolean isEmpty( ) | Returns true if the invoking map is empty. Otherwise, returns false. |
| 5. | void clear( ) | Removes all key/value pairs from the invoking map. |
| 6. | int size( ) | Returns the number of key-value mappings in this map. |

**Table 13: Few Methods of Map interface**

### The Map.Entry Interface

The Map.Entry interface enables us to work with a map entry. The entrySet( ) method declared by the Map interface returns a Set containing the map entries. Each of these set elements is a Map.Entry object. Map.Entry is generic and is declared like this:

```
interface
        Map.Entry<K, V>
```

Where, **K** specifies the type of keys, and **V** specifies the type of values.

| S. No | Methods | Description |
|---|---|---|
| 1. | K getKey( ) | Returns the key corresponding to this entry. |
| 2. | V getValue( ) | Returns the value corresponding to this entry. |
| 3. | void setValue(V value) | Replaces the value corresponding to this entry with the specified value (optional operation). |

**Table 14: Few methods of Map.Entry Interface**

### The Map Classes

Several classes provide implementations of the map interfaces. The classes that can be used for maps are:

### The HashMap Class

The HashMap class extends AbstractMap and implements the Map interface. It uses a hash table to store the map. This allows the execution time of get( ) and put( ) to remain constant even for large sets. HashMap is a generic class that has this declaration:

```
class HashMap<K, V>
```

where, **K** specifies the type of keys, and **V** specifies the type of values.

| S. No | Methods | Description |
|---|---|---|
| 1. | AbstractMap | Implements most of the Map interface. |
| 2. | EnumMap | Extends AbstractMap for use with enum keys. |
| 3. | HashMap | Extends AbstractMap to use a hash table. |
| 4. | TreeMap | Extends AbstractMap to use a tree. |
| 5. | WeakHashMap | Extends AbstractMap to use a hash table with weak keys. |
| 6. | LinkedHashMap | Extends HashMap to allow insertion-order iterations. |
| 7. | IdentityHashMap | Extends AbstractMap and uses reference equality when comparing documents. |

**Table 15: The Map Classes**

| S.No | Constructors | Description |
|---|---|---|
| 1. | HashMap( ) | Constructs a default hash map. |
| 2. | HashMap(Map<? extends K, ? extends V> *m*) | Initializes the hash map by using the elements of *m*. |

**Table 16: Few constructors of HashMap Class**

The following constructors are defined:

The default capacity is 16. The default fill ratio is 0.75.

HashMap does not add any methods of its own. A hash map does not guarantee the order of its elements.

### The TreeMap Class

The TreeMap class extends AbstractMap and implements the NavigableMap interface. It creates maps stored in a tree structure. A TreeMap provides an efficient means of storing key/value pairs in sorted order and allows rapid retrieval. Unlike a hash map, a tree map guarantees that its elements will be sorted in ascending key order. TreeMap is a generic class that has this declaration:

```
class TreeMap<K, V>
```

Where, **K** specifies the type of keys, and **V** specifies the type of values.

The following TreeMap constructors are defined:

| S.No | Constructors | Description |
|---|---|---|
| 1. | TreeMap( ) | Constructs an empty tree map that will be sorted by using the natural order of its keys. |
| 2. | TreeMap(Comparator<? super K> *comp*) | constructs an empty tree-based map that will be sorted by using the **Comparator** *comp*. |
| 3. | TreeMap(Map<? extends K, ? extends V> *m*) | initializes a tree map with the entries from *m*, which will be sorted |
| 4. | TreeMap(SortedMap<K, ? extends V> *sm*) | initializes a tree map with the entries from *sm*, which will be sorted in the same order as *sm*. |

**Table 17: Constructors of TreeMap Class**

TreeMap has no methods beyond those specified by the NavigableMap interface and the AbstractMap class.

### The LinkedHashMap Class

LinkedHashMap extends HashMap. It maintains a linked list of the entries in the map, in the order in which they were inserted. This allows insertion-order iteration over the map. That is, when iterating through a collection-view of a LinkedHashMap, the elements will be returned in the order in which they were inserted.
We can also create a LinkedHashMap that returns its elements in the order in which they were last accessed. LinkedHashMap is a generic class that has this declaration:

```
class LinkedHashMap<K, V>
```

Where, **K**specifies the type of keys, and **V** specifies the type of values.
LinkedHashMap defines the following constructors:

| S.No | Constructors | Description |
|---|---|---|
| 1. | LinkedHashMap( ) | Constructs a default **LinkedHashMap**. |
| 2. | LinkedHashMap(Map<? extends K, ? extends V> *m*) | Initializes the **LinkedHashMap** with the elements from *m*. |
| 3. | LinkedHashMap(int *capacity*) | Initializes the **LinkedHashMap** with the specified capacity. |
| 4. | LinkedHashMap(int *capacity*, float *fillRatio*) | Initializes both capacity and fill ratio. |
| 5. | LinkedHashMap(int *capacity*, float *fillRatio*, boolean *Order*) | Allows us to specify whether the elements will be stored in the linked list by insertion order, or by order of last access. If *Order* is **true**, then access order is used. If *Order* is **false**, then insertion order is used. |

**Table 18: Constructors of LinkedHashMap Class**

The default capacity is 16. The default ratio is 0.75. **Pseudo code**

1. Create a HashMap with keys as Integer object and values as String object.
2. Add the following key-value pairs to hash map: “1 - D”, “2 - B”, “5 - C”, “3 - E”, “4 - A”.
3. Print hash map.
4. Create a LinkedHashMap adding all the entries of hash map.
5. Add entry “7 - F” to linked hash map.
6. Print the linked hash map.
7. Create a tree map adding all entries in linked hash map.
8. Add entry “6 - G” to tree map.
9. Print tree map.

10. Print all entries of tree map as "key : value". For example, 1 : D, 2 : B, .... [Hint: Use Map.Entry interface]

```
                    /* Java program using map classes.*/
import java.util.*;
class MapClasses {

public static void main(String[ ] args) {

// Implement code here (step-1 to step-10 of pseudo code).
 }
}
```

### Problem Validation

```
Input:
N/A
Output:
Contents of hash map:
{1=D, 2=B, 3=E, 4=A, 5=C}
Key Value
1 : D
2 : B
3 : E
4 : A
5 : C
Contents of linked hash map:
 {1=D, 2=B, 3=E, 4=A, 5=C, 7=X, 6=Y}
Contents of tree map:
 {1=D, 2=B, 3=E, 4=A, 5=C}
```

## Program 11

## ENUMERATION AND COMPARATOR

### Problem Definition

A program using Enumeration and Comparator interfaces.

### Problem Description

### The Enumeration Interface

The Enumeration interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects. This legacy interface has been superseded by Iterator.Although not deprecated, Enumeration is considered obsolete for new code. However, it is used by several methods defined by the legacy classes (such as Vector and Properties), it is used by several other API classes, and is currently in widespread use in application code.Because it is still in use, it was retrofitted for generics by JDK 5. It has this declaration:

```
interface Enumeration<E>
        Where E specifies the type of element being enumerated.
```

Enumeration specifies the following two methods:

- boolean hasMoreElements( )
- E nextElement( )

When implemented, hasMoreElements( ) must return true while there are still more elements to extract, and false when all the elements have been enumerated. nextElement( ) returns the next object in the enumeration. That is, each call to nextElement( ) obtains the next object in the enumeration. It throws NoSuchElementException when the enumeration is complete.

### Comparators

Both TreeSet and TreeMap store elements in sorted order. However, it is the comparator that defines precisely what "sorted order" means. By default, these classes store their elements by using what Java refers to as "natural ordering," which is usually the ordering that you would expect (A before B, 1 before 2, and so forth). If you want to order elements a different way, then specify a Comparator when you construct the set or map. Doing so gives you the ability to govern precisely how elements are stored within sorted collections and maps.

Comparator is a generic interface that has this declaration:

```
interface Comparator<T>
Where, T specifies the type of objects being compared.
```

The Comparator interface defines two methods:

- compare( )
- equals( ) compare( ) : The compare( ) method, shown here, compares two elements for order:

```
public int compare(T obj1, T obj2)
```

obj1 and obj2 are the objects to be compared. This method returns zero if the objects are equal. It returns a positive value if obj1 is greater than obj2. Otherwise, a negative value is returned. The method can throw a ClassCastException if the types of the objects are not compatible for comparison. By overriding compare( ), you can alter the way that objects are ordered. For example, to sort in reverse order, you can create a comparator that reverses the outcome of a comparison.

equals( ) : The equals( ) method, shown here, tests whether an object equals the invoking comparator:

```
boolean equals(Object obj)
```

Where, obj is the object to be tested for equality. The method returns true if obj and the invoking object are both Comparator objects and use the same ordering. Otherwise, it returns false. Overriding equals( ) is unnecessary, and most simple comparators will not do so.

**Pseudo code**

1. Define the comparator that arranges objects in descending order.
2. Create a TreeSet of string objects that store elements using the specified comparator.
3. Add elements “D”, “B”, “A”, “C” to tree set.
4. Print the tree set.
5. Create a TreeMap of String keys and Integer values using specified comparator.
6. Add entries “S - 10”, “P - 20”, “R - 30”, “Q - 40”.
7. Print the tree map.
8. Create a Vector that stores strings.
9. Add elements “B”, “D”, “A”, “C”.
10. Get the Enumeration object on vector.

11. Print vector elements using enumerator.

**/* Java program using Enumeration and Comparator interfaces.*/**

```
import java.util.*;
class DescComp implements Comparator<String>{
public int compare(String s1,String s2) {
// Write code here such that when this comparator is used with -
// - TreeSet or TreeMap, there elements are arrange in descending order.
}
}
class EnumAndComparator {
public static void main(String[ ] args) {
// Implement code here (step-2 to step-11 of pseudo code).
}
}
```

**Problem Validation**

```
Input:
N/A
Output:
[D, C, B, A]
{X=10, R=30, Q=40, P=20}
Enumerating through vector elements
B
D
A
C
```

**Program 12**

## FILTER AND BUFFERED I/O STREAMS

### Problem Definition

A program to illustrate the usage of filter and Buffered I/O streams.

### Problem Description

### Filtered streams

Filtered streams are simply wrappers around underlying input or output streams that transparently provide some extended level of functionality. These streams are typically accessed by methods that are expecting a generic stream, which is a superclass of the filtered streams. Typical extensions are buffering, character translation, and raw data translation. The filtered byte streams are FilterInputStream and FilterOutputStream. Their constructors are :

```
FilterOutputStream(OutputStream os)
FilterInputStream(InputStream is)
```

The methods provided in these classes are identical to those in InputStream and OutputStream.


Fig. 11: Classes Hierarchy for FilterInputStream

### Buffered Byte Streams

For the byte-oriented streams, a buffered stream extends a filtered stream class by attaching a memory buffer to the I/O streams. This buffer allows Java to do I/O operations on more than a byte at a time, hence increasing performance. Because the buffer is available, skipping, marking, and resetting of the stream become possible. The buffered byte stream classes are BufferedInputStream and BufferedOutputStream. PushbackInputStream also implements a buffered stream.

### BufferedInputStream

Buffering I/O is a very common performance optimization. Java's BufferedInputStream

class allows you to "wrap" any InputStream into a buffered stream and achieve this performance improvement. BufferedInputStream has two constructors:

| S.No | Constructors | Description |
|---|---|---|
| 1. | BufferedInputStream(InputStream *inputStream*) | creates a buffered stream using a default buffer size. |
| 2. | BufferedInputStream(InputStream *inputStream*, int *bufSize*) | The size of the buffer is passed in *bufSize*. Use of sizes that are multiples of a memory page, a disk block, and so on, can have a significant positive impact on performance. This is, however, implementation-dependent. An optimal buffer size is generally dependent on the host operating system, the amount of memory available, and how the machine is configured. |

**Table 19: Constructors of BufferedInputStream Class**

A good guess for a size is around 8,192 bytes, and attaching even a rather small buffer to an I/O stream is always a good idea. That way, the low-level system can read blocks of data from the disk or network and store the results in your buffer. Thus, even if you are reading the data a byte at a time out of the InputStream, you will be manipulating fast memory most of the time. Buffering an input stream also provides the foundation required to support moving backward in the stream of the available buffer. Beyond the read( ) and skip( ) methods implemented in any InputStream, BufferedInputStream also supports the mark( ) and reset( ) methods. This support is reflected by BufferedInput-Stream.markSupported( ) returning true.

### BufferedOutputStream

A BufferedOutputStream is similar to any OutputStream with the exception of an added flush( ) method that is used to ensure that data buffers are physically written to the actual output device. Since the point of a BufferedOutputStream is to improve performance by reducing the number of times the system actually writes data, you may need to call flush( ) to cause any data that is in the buffer to be immediately written. Unlike buffered input, buffering output does not provide additional functionality. Buffers for output in Java are there to increase performance.

The two available constructors are:

| S.No | Constructors | Description |
|---|---|---|
| 1. | BufferedOutputStream(OutputStream *outputStream*) | Creates a buffered stream using the default buffer size. |
| 2. | BufferedOutputStream(OutputStream *outputStream*, int *bufSize*) | The size of the buffer is passed in *bufSize*. |

**Table 20: Constructors of BufferedOutputStream Class**

### PushbackInputStream

One of the novel uses of buffering is the implementation of pushback. Pushback is used on an input stream to allow a byte to be read and then returned (that is, "pushed back") to the stream. The PushbackInputStream class implements this idea. It provides a mechanism to "peek" at what is coming from an input stream without disrupting it.

PushbackInputStream has the following constructors:

| S.No | Constructors | Description |
|---|---|---|
| 1. | PushbackInputStream(InputStream *inputStream*) | Creates a stream object that allows one byte to be returned to the input stream. |
| 2. | PushbackInputStream(InputStream *inputStream*, int *numBytes*) | Creates a stream that has a pushback buffer that is *numBytes* long. This allows multiple bytes to be returned to the input stream. Beyond the familiar methods of InputStream, PushbackInputStream provides unread( ). |

**Table 21: Constructors of PushbackInputStream Class**

PushbackInputStream unread( ) Method :

| S.No | Constructors | Description |
|---|---|---|
| 1. | void unread(int *ch*) | pushes back the low-order byte of *ch*. This will be the next byte returned by a subsequent call to **read( )**. |
| 2. | void unread(byte *buffer*[ ]) | It returns the bytes in *buffer*. |
| 3. | void unread(byte *buffer*, int *offset*, int *numChars*) | Pushes back *numChars* bytes beginning at *offset* from *buffer*. An **IOException** will be thrown if there is an attempt to return a byte when the pushback buffer is full. |

**Table 22: Overloaded versions of unread( ) method**

### Pseudo code

### BufferedInputStream

1. Create a String object containing text as “This is a © copyright symbol but this is &copy not.”
2. Convert above string to byte[ ].
3. Create a ByteArrayInputStream to read from the byte[ ].
4. Wrap ByteArrayInputStream into a BufferedInputStream.
5. Read bytes using BufferedInputStream.
6. If byte sequence is © then
7. Print the copyright symbol (c) instead of © .
8. Otherwise, print the bytes as it is.

### PushbackInputStream

1. Create a String object containing text as

```
"if (a == 4) a = 0;\n".
```

2. Convert above string to byte[ ].
3. Create a ByteArrayInputStream to read from the byte[ ].
4. Wrap ByteArrayInputStream into a PushbackInputStream.
5. Read bytes using PushbackInputStream.
6. If byte sequence is "= =" then print ".eq.".
7. Else if byte is "=" then print "$¡-$".
8. else, print the bytes as it is.

**BufferedInputStream**

**/* Demonstrate BufferedInputStream.*/**

```
import java.io.*;
class BufferedInputStreamDemo {

   public static void main(String args[ ]) throws IOException {

String s ="This is a &copy; copyright symbol "+"
              but this is &copy not.\n";
      // Implement pseudo code from step-2 to step-8 here.
   }
}
```

**PushbackInputStream**

**/*Demonstrate PushbackInputStream. */**

```
import java.io.*;
class PushbackInputStreamDemo {
  public static void main(String args[ ]) throws IOException {
String s = "if (a == 4) a = 0;\n";
// Implement pseudo code from step-2 to step-8 here.
    }
}
```

**Problem Validation**

```
Input:
BufferedInputStream
"This is a &copy; copyright symbol "+"but this is &copy not.\n";
PushbackInputStream
"if (a == 4) a = 0;\n";
Output:
BufferedInputStream
This is a (c) copyright symbol but this is &copy not.
PushbackInputStream
if (a .eq. 4) a <- 0;
```

## Program 13

## SERIALIZATION

### Problem Definition

A program to illustrate the usage of Serialization.

### Problem Description

### Serialization

Serialization is the process of writing the state of an object to a byte stream. This is useful when you want to save the state of some objects to a persistent storage area, such as a file. At a later time, you may restore these objects by using the process of deserialization.

Serialization is also needed to implement Remote Method Invocation (RMI). RMI allows a Java object on one machine to invoke a method of a Java object on a different machine. An object may be supplied as an argument to that remote method. The sending machine serializes the object and transmits it. The receiving machine deserializes it.

### Serializable

Only an object that implements the Serializable interface can be saved and restored by the serialization facilities. The Serializable interface defines no members. It is simply used to indicate that a class may be serialized. If a class is serializable, all of its subclasses are also serializable. Variables that are declared as transient are not saved by the serialization facilities. Also, static variables are not saved.

### ObjectOutputStream

The ObjectOutputStream class extends the OutputStream class and implements the ObjectOutput interface. It is responsible for writing objects to a stream. A constructor of this class is

```
ObjectOutputStream(OutputStream outStream) throws IOException
```

The argument outStream is the output stream to which serialized objects will be written. The writeObject( ) method of this class is used to serialize an object. This method can throw an IOException on error condition.

```
public final void writeObject(Object obj):-
Write the specified object to the ObjectOutputStream.
```

### ObjectInputStream

The ObjectInputStream class extends the InputStream class and implements the Object-Input interface.

ObjectInputStream is responsible for reading objects from a stream. A constructor of this class is

```
ObjectInputStream(InputStream inStream) throws IOException
```

The argument inStream is the input stream from which serialized objects should be read.

The readObject( ) method this class is used to deserialize an object. This method can throw either an IOException on error condition or it can throw ClassNotFoundException.

```
public final Object readObject()
throws IOException,ClassNotFoundException :-
Read an object from the ObjectInputStream.
```

**Pseudo code**

1. Create an object of Student class.
2. Create a FileOutputStream object. Connect it to file "student.ser".
3. Print student object before serialization.
4. Wrap FileOutputStream object into ObjectOutputStream.
5. Serialize the student object.
6. Flush and close the output streams.
7. Create a FileInputStream object. Connect it to same file "student.ser".
8. Wrap FileInputStream object into ObjectInputStream.
9. Deserialize and print the sudent object.
10. Close the input streams.

**/* Java program to illustrate serialization. */**

```
import java.io.*;

class Student implements Serializable {

int rno =100;
String name;
float attd;

Student(int rno,String name,float attd) {

this.rno = rno;
this.name = name;
this.attd = attd;
```

```
}
public String toString( ) {

return "{ Roll No : "+rno+", Name : "+name+", Attendance : "+attd+" }";
}
}
class SerializationDemo {

public static void main(String[ ] args) throws IOException,
                                        ClassNotFoundException{

Student s1 = new Student(1,"ABC",82.25f);

           // Implement Serialization & De-serialization code here -
          // - (step-2 to step-10 of pseudo code).
     }
}
```

### Problem Validation

```
Input:
N/A
Output:
State of object before serialization
{ Roll No : 1, Name : ABC, Attendance : 82.25 }
State of object after deserialization
{ Roll No : 1, Name : ABC, Attendance : 82.25 }
```

**Program 14**

## GUI WITH DIFFERENT CONTROLS

### MENUS AND EVENT HANDLING

**Problem Definition**

An application involving GUI with different controls, menus and event handling.

**Problem Description**

**Control Fundamentals**

The AWT supports the following types of controls:

- Labels
- Push buttons
- Check boxes
- Choice lists
- Lists
- Scroll bars
- Text editing

These controls are subclasses of java.awt.Component class.
Adding and Removing Controls
To include a control in a window, you must add it to the window. To do this, you must first create an instance of the desired control and then add it to a window by calling add( ), which is defined by Container. The add( ) method has several forms. The following form is the one that is used for the first part of this chapter:

```
Component add(Component compObj)
```

Here,compObj is an instance of the control that you want to add. A reference to compObj is returned. Once a control has been added, it will automatically be visible whenever its parent window is displayed. Sometimes you will want to remove a control from a window when the control is no longer needed. To do this, call remove( ). This method is also defined by Container.

It has this general form:

```
void remove(Component obj)
```

Here, obj is a reference to the control you want to remove. You can remove all controls by calling removeAll( ).

**Pseudo code**

**GUI Controls**

1. Create all the Components (task is to implement createComps( ) method).
   - Create all the Label objects.
   - Create all TextField objects (tf_Name, tf_Rno) with size of 15 characters.
   - Create Checkbox objects (cb_Male, cb_Female). Combine them to Checkbox-Group "gen".
   - Create Checkbox objects (cb_Eng, cb_Hin, cb_Tel, cb_Urdu).
   - Create Choice object (ch_Course). Add the following course items to it "B.E","B.Pharm","B.Sc", "M.B.A", "M.C.A".
   - Create List object (ls_Hob). Add the following items to it "Watching T.V", "Playing Sports", "Listening Music", "Surfing Internet".
   - Create Scrollbar object (sb_Skill). Orient scrollbar horizontally, set its initial value to 5, set thumb size to 1, set minimum value to 0 and set maximum value to 11.
   - Create Button object (submit).
2. Set properties for the component.
3. Add all Components in the required order to applet (refer output).
4. Register Listener for the submit button. The listener object is the same object (source object).
5. Extract the details entered in the form and display (refer output).
   - Extract the data from all components and get their values into string objects s_name, s_rno, s_gen etc.
   - Call repaint( ) method to update the applet window.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="StudentForm" width=425 height=600 >
</applet>
*/

public class StudentForm extends Applet implements ActionListener {
String s_name, s_rno, s_course, s_addr, s_gender, s_hob,
                s_skillLevel, s_lang;
Label lb_Header, lb_Name, lb_Rno, lb_Course, lb_Gender,
              lb_Langs, lb_Hob,lb_SkillLevel, lb_Addr;
TextField tf_Name,tf_Rno;
TextArea ta_Addr;
```

```
CheckboxGroup gen;
Checkbox cb_Male, cb_Female, cb_Eng, cb_Hin, cb_Tel, cb_Urdu;
Choice ch_Course;
List ls_Hob;
Scrollbar sb_SkillLevel;
Button submit;
public void init( ) {
createComps( );
setCompProperties( );
addComps( );
registerListener( );
}
```

**/*Method for Creating All Components*/**

```
public void createComps( ) {
s_name=s_rno=s_course=s_addr=s_gender=s_hob=s_skillLevel=s_lang="";
// create all components here (i.e., implement step-1 of pseudo code).
}
```

**/*Method for setting Components properties.*/**

```
public void setCompProperties( ) {
setBackground(new Color(120,120,150));
setForeground(Color.black);
lb_Header = new Label(" STUDENT REGISTRATION FORM ",Label.CENTER);
lb_Header.setFont(new Font(Font.SERIF,Font.BOLD,25));
lb_Name.setFont(new Font(Font.SERIF,Font.BOLD,14));
lb_Rno.setFont(new Font(Font.SERIF,Font.BOLD,14));
lb_Course.setFont(new Font(Font.SERIF,Font.BOLD,14));
lb_Gender.setFont(new Font(Font.SERIF,Font.BOLD,14));
lb_Langs.setFont(new Font(Font.SERIF,Font.BOLD,14));
lb_Hob.setFont(new Font(Font.SERIF,Font.BOLD,14));
lb_SkillLevel.setFont(new Font(Font.SERIF,Font.BOLD,14));
lb_Addr.setFont(new Font(Font.SERIF,Font.BOLD,14));
sb_SkillLevel.setBackground(Color.GREEN);
sb_SkillLevel.setForeground(Color.RED);
sb_SkillLevel.setPreferredSize(new Dimension(400,20));
}
\begin{verbatim}
/*Method for adding All Components to applet window.*/

\begin{verbatim}
public void addComps( ) {
// Add all components here. (i.e., step-3 of pseudocode).
}
```

**/*Method for registering listener object for button Component.*/**

```
public void registerListener( ) {
// register listener object for submit button
                    (i.e., step-4 of pseudocode).
}
public void paint(Graphics g) {
Font myFont = new Font("TIMESROMAN",Font.BOLD,18);
g.setFont(myFont);
g.drawString("Your Details Are : ",100,380);
myFont = new Font("TIMESROMAN",Font.PLAIN,14);
g.setFont(myFont);
g.drawString(s_name,10,400);
g.drawString(s_rno,10,420);
g.drawString(s_gender,10,440);
g.drawString(s_addr,10,460);
g.drawString(s_lang,10,480);
g.drawString(s_course,10,500);
g.drawString(s_hob,10,520);
g.drawString(s_skillLevel,10,540);
}
```

**/*Method for handling event raise on submit button (when submit button is clicked).*/**

```
public void actionPerformed(ActionEvent ae) {
if(ae.getSource().equals(submit)) {
// extract data from all components and store values in string objects.
// step-5 of pseudocode.
}
// call repaint here.
}
}
```

**Menu Bar**

1. Create a MenuBar object (mb).
2. Create Menu objects (editOptions and bgColor).
3. Create MenuItem objects (redColor, greenColor, blueColor).
4. Create CheckboxMenuItem object (wordWrap).
5. Add all MenuItem, CheckboxMenuItem and Menu obects.

- Add MenuItem objects (redColor, greenColor and blueColor) to Menu object (bgColor).
- Add Menu objects (bgColor and wordWrap) to Menu object (editOptions).
- Add Menu object (editOptions) to MenuBar object (mb).
- Set MenuBar object (mb) to frame.

6. Register listener for MenuItem objects (redColor, greenColor and blueColor). Listener is the same object (source object).
7. Register listener for CheckboxMenuItem object (wordWrap). Listener is the same object (source object).
8. Handle events for MenuItem objects (redColor, greenColor and blueColor). Based on the MenuItem clicked set the background color of the frame to that color.
9. Handle events for MenuItem objects (wordWrap). Based on wordWrap turned on/off update the title of the frame with appropriate message.

**/* Program to Demonstrate MenuBar in Java.*/**

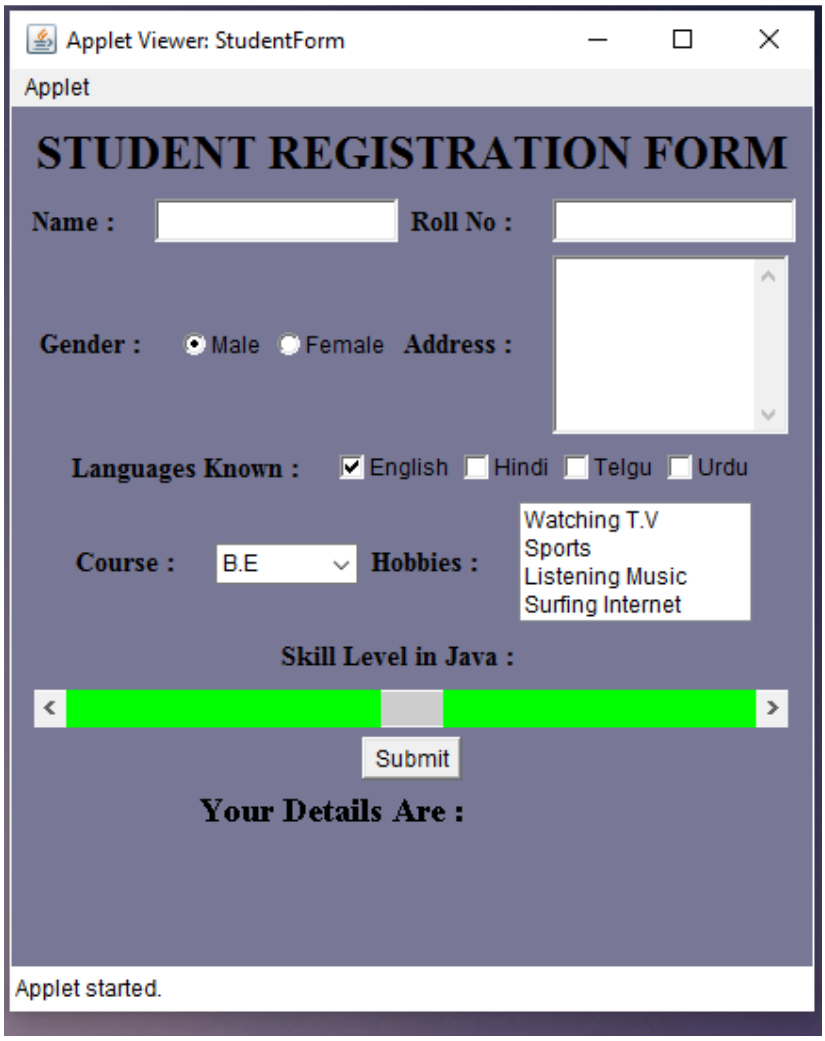
```
import java.awt.*;
import java.awt.event.*;

class MenuBarDemo extends Frame implements ActionListener, ItemListener {

MenuBar mb;
Menu editOptions, bgColor;
MenuItem redColor, greenColor, blueColor;
CheckboxMenuItem resize;

MenuBarDemo( ) {

createMenus( );
addMenuItems();
registerListener( );
setResizable(false);
setSize(500,500);
setVisible(true);
setTitle("Menus Demo");
}
void createMenus( ) {

// implement step-1 to step-4 of pseudo code here.

}
void addMenuItems( ) {
```

```
// Add all MenuItem, CheckboxMenuItem and Menu objects
// (Step-5 of pseudocode).
}
void registerListener( ) {

// Register listener for MenuItem and CheckboxMenuItem objects
// (Step-6 and Step-7 of pseudocode).
}
public void actionPerformed(ActionEvent ae) {

// events for MenuItem objects (redColor, greenColor and blueColor).
// If button \Red" is clicked then set background color of frame
    to red and
// so on (step-8 of pseudo code) .

}
public void itemStateChanged(ItemEvent ie) {

// Handle events for MenuItem objects (resize).
// if ""Is Resizable" is turned on then set resizable property of
     frame to true.
// Otherwise set resizable property of frame to false
// step-9 of pseudocode.
}
public static void main(String[ ] args) {
new MenuBarDemo();
}
}
```

**Problem Validation**

**GUI Controls**

```
Input:
N/A
Output:
```




Screen Shot 1: Registration Form initial state

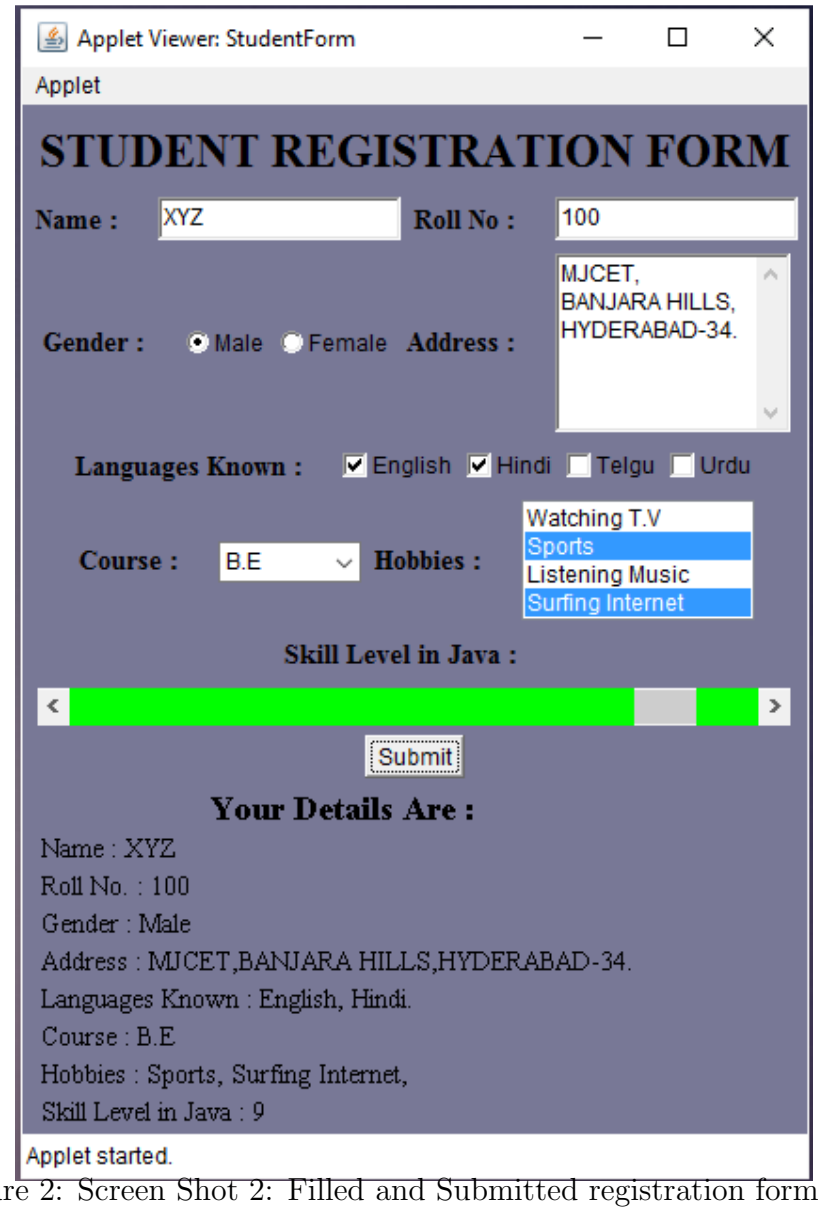



Figure 2: Screen Shot 2: Filled and Submitted registration form

Menus
Input:
N/A
Output:

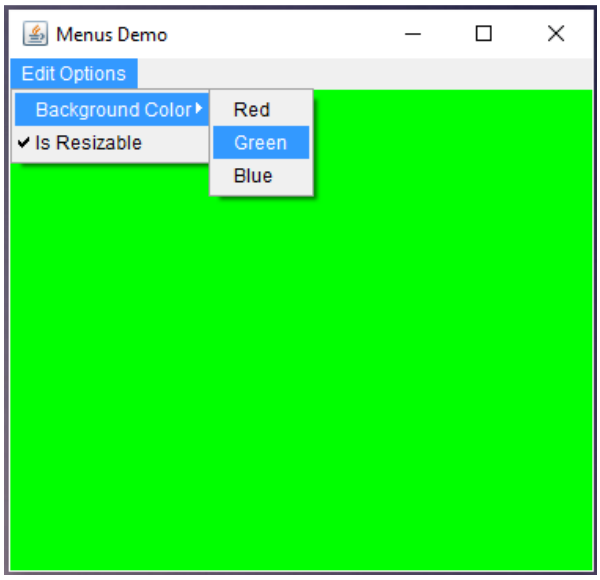



Figure 3: Screen Shot 3: On Clicking MenutItem green

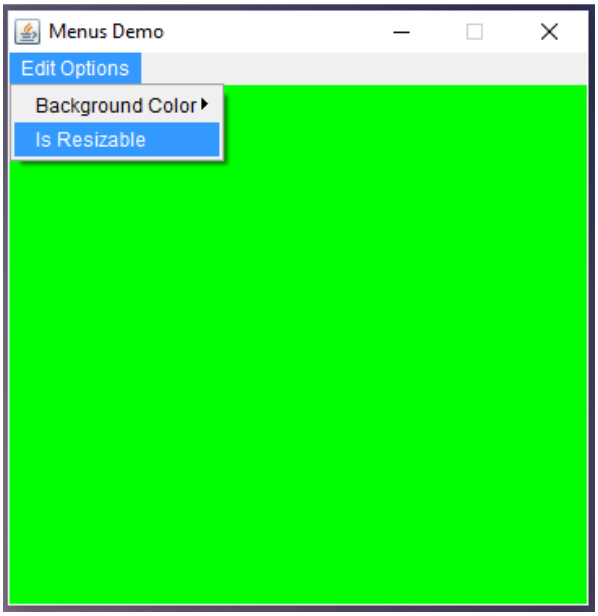



Figure 4: Screen Shot 4: Is Resizable is disabled on Frame

## Program 15

## AWT/SWING

### Problem Definition

A program to implement AWT/Swing.

### Problem Description

The Swing toolkit includes a rich set of components for building GUIs and adding interactivity to Java applications. Swing includes all the components you would expect from a modern toolkit: table controls, list controls, tree controls, buttons, and labels.

Swing is far from a simple component toolkit. However, It includes rich undo support, a highly customizable text package, integrated internationalization and accessibility support. To truly leverage the cross-platform capabilities of the Java platform, Swing supports numerous look and feels, including the ability to create your own look and feel. The ability to create a custom look and feel is made easier with Synth, a look and feel specifically designed to be customized. Swing wouldn't be a component toolkit without the basic user interface primitives such as drag and drop, event handling, customizable painting, and window management.

Swing is part of the Java Foundation Classes (JFC). The JFC also include other features important to a GUI program, such as the ability to add rich graphics functionality and the ability to create a program that can work in different languages and by users with different input devices.

### Pseudocode

1. Create Components.
    - Create three JButton objects b1, b2 and b3.
    - To get image icons on buttons, store the gif image files in directory "images". Directory "images" must be the sub directory of current working directory.
2. Set properties for the component.
3. Add the button Components to JPanel.
4. Register Listener for the submit button.
5. Handle events on button.

```
import javax.swing.AbstractButton;
import javax.swing.JButton;
import javax.swing.JPanel;
import javax.swing.JFrame;
import javax.swing.ImageIcon;
```

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;

/*
 * ButtonDemo.java requires the following files:
 *   images/right.gif
 *   images/middle.gif
 *   images/left.gif
 */
public class ButtonDemo extends JPanel
                        implements ActionListener {

protected JButton b1, b2, b3;
ImageIcon leftButtonIcon,  middleButtonIcon, rightButtonIcon;

    public ButtonDemo( ) {

 leftButtonIcon = createImageIcon("images/right.gif");
         middleButtonIcon = createImageIcon("images/middle.gif");
       rightButtonIcon = createImageIcon("images/left.gif");

createButtons( );
setProperties( );
addButtons( );
registerListeners( );
    }

public void createButtons( ) {
// Create Buttons b1,b2,b3 containing images (step-1 of pseudo code).
}
public void setProperties( ) {

      b1.setVerticalTextPosition(AbstractButton.CENTER);
        b1.setHorizontalTextPosition(AbstractButton.LEADING);
        b1.setMnemonic(KeyEvent.VK_D);
        b1.setActionCommand("disable");

        b2.setVerticalTextPosition(AbstractButton.BOTTOM);
        b2.setHorizontalTextPosition(AbstractButton.CENTER);
        b2.setMnemonic(KeyEvent.VK_M);

b3.setMnemonic(KeyEvent.VK_E);
b3.setActionCommand("enable");
b3.setEnabled(false);
b1.setToolTipText("Click this button to disable the middle button.");
```

```
b2.setToolTipText("This middle button does nothing when you click it.");
b3.setToolTipText("Click this button to enable the middle button.");
}
```

**/* Add Components to this container, using the default FlowLayout. */**

```
public void addButtons() {

// Add buttons here (step-3 of pseudo code).
}
```

**/* Listen for actions on buttons 1 and 3. */**

```
public void registerListeners() {

// Register listeners here (step-4 of pseudo code).
}

public void actionPerformed(ActionEvent e) {

// Implement event handling code here (step-5 of pseudo code).
     }

    /* Returns an ImageIcon, or null if the path was invalid. */

    protected static ImageIcon createImageIcon(String path) {

java.net.URL imgURL = ButtonDemo.class.getResource(path);
         if (imgURL != null) {
             return new ImageIcon(imgURL);
}
 else {
             System.err.println("Couldn't find file: " + path);
             return null;
         }
    }
```

**/*Create the Frame and display it. For thread safety,this method should be invoked from the event-dispatching thread.*/**

```
    private static void createAndShowGUI( ) {

    JFrame frame = new JFrame("ButtonDemo");
     //Create and set up the window.
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
     ButtonDemo contentPane = new ButtonDemo( );
     /* Create and set up the content pane. */

     contentPane.setOpaque(true); /* Content panes must be opaque. */
     frame.setContentPane(contentPane);
     frame.pack( );     /* Display the window the required size. */
     frame.setVisible(true);
    }
```

**/* Schedule a job for the event-dispatching thread:*/**

**creating and showing this application's GUI.**

```
    public static void main(String[ ] args) {
        javax.swing.SwingUtilities.invokeLater(new Runnable( ) {
            public void run( ) {
                createAndShowGUI( );
            }
        });
    }
}
```

**Problem Validation**

```
Input:
N/A
Output:
```


Figure 5: Screen Shot 5: Middle button enabled

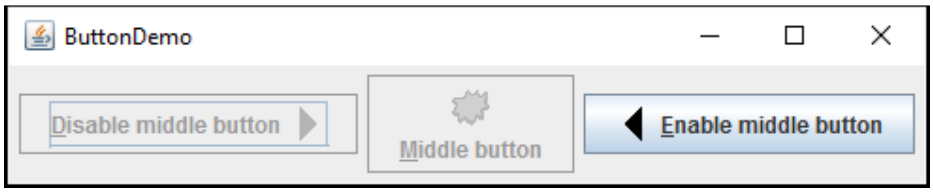



Figure 6: Screen Shot 6: Middle button disabled

## Annexure – I

**List of programs according to O.U. curriculum**

**CS 281**

### JAVA LAB

```
Instruction                     3 Periods per week
Duration of University Examination 3 Hours
University Examination            50 Marks
Sessional                      25 Marks
```

1. A program to illustrate the concept of class with constructors, methods and overloading.
2. A program to illustrate the concept of inheritance and dynamic polymorphism.
3. A program to illustrate the usage of abstract class.
4. A program to illustrate multithreading.
5. A program to illustrate thread synchronization.
6. A program using StringTokenizer.
7. A program using Linked list class.
8. A program using TreeSet class.
9. A program using HashSet and Iterator classes.
10. A program using map classes.
11. A program using Enumeration and Comparator interfaces.
12. A program to illustrate the usage of filter and Buffered I/O streams.
13. A program to illustrate the usage of Serialization.
14. An application involving GUI with different controls, menus and event handling.
15. A program to implement AWT/Swing.

**Suggested Reading:**

1. Herbert Schildt, The Complete Reference Java, 7th Edition, Tata McGraw Hill, 2005.

2. James M Slack, Programming and Problem solving with JAVA, Thomson Learning, 2002

3. C Thomas Wu, An Introduction to Object Oriented programming with Java, Tata McGraw Hill, 2005.

**References:**

1. The Java™ Tutorials https://docs.oracle.com/javase/tutorial/.

2. Herbert Schildt, Java : The Complete Reference, Seventh Edition, McGraw-Hill Companies, Inc.

3. *java*™ Platform, Standard Edition 6 API Specification, http://docs.oracle.com/javase/6/docs/api/