INDEX

Preface 4

Acknowledgements 6

CHAPTER-1 INTRODUCTION TO JAVA CONCEPTS

1.1 History of java 7
  1.1.1 Characteristics of Java 7

1.2 Object Oriented Programming 7
  1.2.1 Definition of Object 8
  1.2.2 JVM 8
  1.2.3 The Byte code 9
  1.2.4 JDK 9
  1.2.5 JRE 9
  1.2.6 OOP Principles: Encapsulation, Inheritance and Polymorphism. 10
1.3 JAVA as a OOP 11
1.4 JAVA is an Internet Enabled Language 11
1.5 Class Definition 12
1.6 A simple Java Program 12
1.7 JAVA Variables 13
1.8 JAVA Keywords 14
1.9 Identifiers 15
1.10 Operators 15
1.11 Control statements 15
1.12 Data types 16
1.13 JAVA API 19
1.14 Arrays 20
1.15 The Concept of Classes and Objects 21
1.16 Object 22
1.17 Assigning Object Reference Variables 23
1.18 JAVA Method 24
1.19 Constructors 25
1.20 Access Control 28
1.21 Static Methods 30
1.22 Usage of Final with Data 31
1.23 Command Line Arguments in Java Program 32
1.24 Overloading Constructors 32
1.25 Overloading Methods 34
1.26 Parameter Passing – Call by Value 35
1.27 Nested and Inner Classes 37
1.28 Exploring the String Class 37

CHAPTER-2 INHERITANCE, PACKAGES, INTERFACES AND I/O STREAMS

2.1 Inheritance 40
- 2.1.1 Forms of Inheritance 41
- 2.1.2 Usage of super keyword 43

2.2 Method Overriding 44
2.3 Abstract Class 45
2.4 Dynamic Method Dispatch 47
2.5 Using Final with Inheritance 48
2.6 Object Class 49
2.7 Packages 49
- 2.7.1 Scope of the Variables 50

2.8 Interfaces 52
- 2.8.1 Extending Interfaces 53

2.9 I/O Streams 55
2.9.1 Stream Classes 56
57

CHAPTER-3 EXCEPTION HANDLING AND MULTITHREADING

3.1 Fundamentals of Exception Handling 63
3.1.1 Types of Exceptions 63
3.2 Usage of Try and Catch 64
- 3.2.1 Multiple Catch Blocks 65

3.3 Generic Exception 67
3.4 Throw, Throws and Finally Keywords 69
3.5 Java's Built-in Exceptions 72
3.6 Concepts of Multithreading 75
- 3.6.1 Multithreading concept 75
- 3.6.2 Thread Life Cycle 77

3.7 Creating Threads 77
- 3.7.1 Extending a class 78
- 3.7.2 Implementing an interface 78

3.8 Creating Multiple Threads using Thread class 80
3.9 Methods Defined in Thread Class 81
3.10 Thread Priorities 81
3.11 Synchronization 83
3.12 Interthread Communication 85
3.13 Deadlocks 88
3.14 Thread groups 89

CHAPTER-4 APPLETS, EVENT HANDLING AND SWINGS

4.1 Applets 91
4.2 Applet Structure and Elements 91
4.3 Applet Class 92
4.4 Methods of java applet 93
4.5 Parameter Passing to Applets 95
4.6 Event Handling 97
4.7 The Delegation Event Model 98

4.8 Event Classes 98
4.9 The Delegation Event Model working 100
4.10 Event Sources 101
4.11 The Event Listeners Interfaces 102
4.12 Adding and removing controls 106
4.13 AWT Graphics 111
4.14 Swings–Introduction 114

CHAPTERS-5 INTRODUCTION TO JAVA DATABASE CONNECTIVITY

5.1 What is JDBC? 124
5.2 Driver Types 124
5.3 Establishing Connection to Database 126
5.4 Creating JDBC Statements 127
5.5 JDBC API 128
5.6 Example Program using JDBC API 128-132

6.ADDITIONALS

6.1. Java interview Questions 133-135
6.2. Lab Manual covering 5 chapters with questions & exercises at the end of each program. 136-184

Preface

Java is an object oriented programming language, revolutionized the software industry being an open soure, developing software for, right from small gadgets to business, web and network and mobile etc.

It has become popular, as it is the first open soure object oriented language used to develop any kind of application and particular web applications and applet is considered as one of its best feature.

Object oriented concepts understanding is must for doing java programming easily. Mastering of APIs, the way they developed to use it in java programming makes one can be master in java language.

This book is written exclusively from student dimension after taking into consideration of difficulties and problems of learning in a given semester with in a limited time. The Book is to master fundamentals and some important basic insights of APIs to do program easily. The Students enthusiasm while I was teaching in the class room and learning interest in the laboratory and the final results were the sources of motivation to write this book.

I taught JAVA several times for UG and PG students is of more practical than theory. Every topic is covered in terms of real world simple application is really joy of learning by the students.

My immense interest in this subject comes as I had worked in Singapore for 3 years in java technologies and also java certified in 2001 in Singapore and Microsoft Certified in 1999 in India.

This book is covering core Java topics covering 5 chapters; syllabus covered is most common to all universites in AP a part from KL University. This book can be useful to PG Computer Science students and other branches of study who wants to learn java apart from all Computer Science related branches.

Coverage of content is planned to practice and learn possibly in a semester time.One can master fundamentals, APIs and sufficient practice in laboratory with examples covered in a well defined manner.

The book is written with an idea in mind that what ever concept/ topic lisetend in class room the same they do it in laboratory immediately with real world common sense application, is one of the most essential feature of this book and so it is titled “Core Java”. And this book contains important placement questions which were used by my earlier stuents and benefited a lot.

Finally, a word to say that student can pick up this book; straight a way can go through reading & understanding concepts and practice them in lab with prior basic knowledge of programming knowledge in C or C++.

Acknowledments

Foremost inner inspiration and driving force of writing this book is our Honourable Chancellor, **Er. Koneru Satyanaryana**, KL University. Also thankful to my senior Colleague, Professor Dr. K. Rajashekara Rao, Dean, Student & Faculty affairs, KL University.

I would like to express my sincere thanks to all those motivated and helped in preparing content topic wise of this book directely and indirectly. I am very much thankful to V.Sangeetha, Nazeer.SK, Research Scholors and Krishna Kumari, Teaching Assistant, KL University, who did great job in ordering the content and preparation of text and images design. And particularly, some of my students of CSE & MCA, KL University.

Author

CHAPTER-1

INTRODUCTION TO JAVA CONCEPTS

1.1History of Java

The Java programming language was originally called Oak, and was designed for use in embedded consumer-electronic applications by James Gosling in 1995. The Java programming language is a general-purpose concurrent class-based object-oriented programming language, specifically designed to have as few implementation dependencies as possible. It allows application developers to write a program once and then be able to run it everywhere on the Internet.

1.1.1 Characteristics of Java

- Simple,
- Object-oriented,
- Distributed,
- Interpreted,
- Robust,
- Secure,
- Architecture-neutral,
- Portable,
- Multithreaded and
- Dynamic

1.2Object Oriented Programming

Object-oriented programming (oop) is a programming paradigm that uses "objects"-data structures consisting of data fields and methods together with their interactions-to design applications and computer programs. All the java programs are object-oriented.

Object Oriented Programming techniques include features such as data abstraction, encapsulation, modularity, polymorphism, and inheritance.

Benefits of object oriented programming

- Reusability (reusable components)
- Reliability
- Robustness
- Extensibility
- Maintainability

1.2.1 Definition of Object

An **object** is any entity that can be manipulated by the commands of a programming language, such as a value, variable and function.

Software objects model real-world objects or abstract concepts such as dog, bicycle, and queue.
Real world objects have states and behaviors.

For example: Dogs states: name, color, and breed, hungry.
Dogs behaviors: barking, fetching.

How do Software objects implement real-world objects?

- Use variables to implement states
- Use methods to implement behaviors

An object is a software bundle of variables and related methods

1.2.2 Java Virtual Machine (JVM)

JVM is the main component of java architecture and it is the part of the JRE (java runtime environment). It provides the cross platform functionality to java. This is a software process that converts the compiled java byte code to machine code.

Byte code is an intermediary language between java source and the host system. Most programming language like C and Pascal converts the source code to the machine code for one specific type of machine as the machine language vary from system to system. Mostly compiler produces code for a particular system but java compiler produce code for virtual machine. JVM provides security to java.

The programs written in Java (or) the source code translated by java compiler into byte code and after that the JVM converts the byte code into machine code for the computer.Byte code is the compiled format for java programs. Once a java program has been converted to byte code, it can be transferred across a network and executed by Java Virtual Machine (JVM).

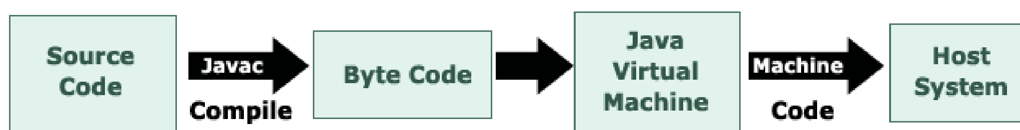


Fig 1.1 JVM Structure

1.2.3 The Byte Code

The Java compiler translates Java source code into Java Byte code is a pseudo-machine language. Byte code is executed by the Java Virtual Machine

(JVM). Byte code is computer object code that is processed by a program. It usually referred to as a virtual machine, rather than by the "real" computer machine, the hardware processor. The virtual machine converts each generalized machine instruction into a specific machine instruction or instructions that this computer's processor will understand. Byte code is the result of compiling source code written in a language that supports this approach

1.2.4 JDK (Java Development Kit)

Java Developer Kit contains tools needed to develop the Java programs, and JRE to run the programs. The tools include compiler (javac.exe), Java application launcher (java.exe), Applet viewer, etc,.

Java Compiler converts java code into byte code. Java application launcher opens a JRE, loads the class, and invokes its main method. You need JDK, if at all you want to write your own programs, and to compile them. For running java programs, JRE is sufficient. JRE is targeted for execution of Java files. JRE(Java Runtime Environment) = JVM + Java Package Classes (like util, math, lang, awt, swing etc,. which will discussed in further chapters) and runtime libraries.

JDK is mainly targeted for java development. i.e. you can create a Java file (with the help of Java packages), compile a Java file and run a java file.

1.2.5 JRE (Java Runtime Environment)

Java Runtime Environment contains JVM, class libraries, and other supporting files. It does not contain any development tools such as compiler, debugger, etc. Actually JVM runs the program, and it uses the class libraries, and other supporting files provided in JRE. If you want to run any java program, you need to have JRE installed in the system.

The java virtual machine provides a platform-independent way of executing code; programmers can concentrate on writing software, without having to be concerned with how or where it will run. If You just want to run applets (say for example Online Yahoo games or puzzles), *JRE* needs to be installed on the machine.

Some JVMs support **just-in-time compilation (JIT)**.Translate byte code instructions into machine language when they are first encountered (byte code into executable code such that .class file to .exe file)

1.2.6 OOP Principles

There are three principles to be implemented by any object oriented programming language. They are:

1. Encapsulation
2. Polymorphism
3. Inheritance

Encapsulation Definition

Encapsulation conceals the functional details of a class from objects that send messages to it. Encapsulation means that the internal representation of an object is generally hidden from view outside of the object's definition.

Example

The Dog is barking. How does it exactly a bark happens (e.g., by inhale and than exhale of dog at a particular pitch and volume it happens).

Think of encapsulation as a black box; data is sent to a method, a lot of work goes on using the data, of which you don't know (or) care about. An output is returned to the caller. That is the process of encapsulation, or information hiding. (It is called information hiding by encapsulation)

Inheritance Definition

Inheritance is a way to form new classes using classes that have already been defined. Object-oriented programming allows classes to inherit commonly used state and behavior from other classes. It is a process by which one object acquires properties of another object. In below figure shown that Bicycle now becomes the super class of MountainBike, RoadBike, and TandemBike. In Java programming language, each class is allowed to have one direct superclass, and each superclass has the potential for an unlimited number of subclasses.

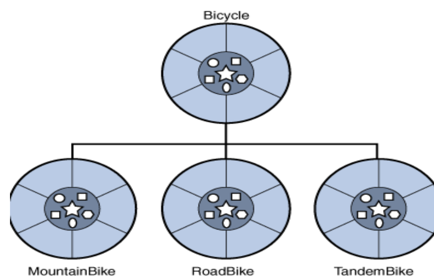


Fig 1. 2 Example of inheritance

Polymorphism Definition

Polymorphism is to allow an entity such as a variable, a function, or an object to have more than one form. In object-oriented programming, polymorphism (from the Greek meaning "having multiple forms") expressed by one interface, multiple methods.

Example: Dog's sense of smell is polymorphic

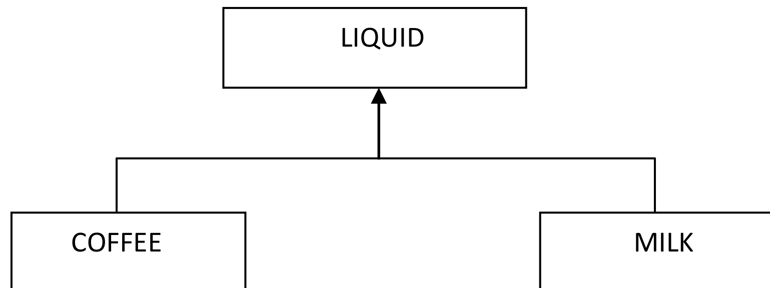


Fig 1.3 Example of Polymorphism

In the above example coffee is a liquid and milk is a liquid.

1.3 Java as an OOP

Java is an object oriented programming and to understand the functionality of OOP in Java, we first need to understand several fundamentals related to objects. These include class, method, inheritance, encapsulation, abstraction, polymorphism etc. which are briefly explained above.

- Objects are more "real-world".
- Objects provide the flexibility and control necessary to deal with evolving requirements.
- Object use facilitates collaboration.
- Objects help manage complexity.
- Reusability, maintainability and extensibility.

1.4 JAVA is an Internet Enabled Language

Java became a general purpose language that had many features to support it as the internet language. Few of the features that favor it to be an internet language are:

- Cross Platform Compatibility
- Support to Internet Protocols
- Support to HTML
- Support to Java Reflection APIs
- Support to XML parsing
- Support to Web Services
- Support to java enabled Mobile devices
- Support to Personal Digital Assistants

1.5 Class Definition

In object oriented programming, a class is a construct that is used as a blueprint (or template) to create objects of that class. This blueprint describes the state and behavior that the objects of the class all share.

A Java class is a group of java methods and variables .A class can be broken down into two things:

i) The first piece is a *class-header* which consists of the keyword "class" and the name you give to the class.

ii) The second piece is the body. It consists of a pair of open/close squiggly brackets with stuff in between them.

Syntax

```
class class_name
{
  type instance_variable1;
  type instance_variable2;

 type method1(parameter_list){
  }
}
```

An object of a given class is called an instance of the class. The class that contains (and was used to create) that instance can be considered as the type of that object, e.g. an object instance of the "Fruit" class would be of the type "Fruit".

A class usually represents a noun, such as a person, place or (possibly quite abstract) thing – it is a model of a concept within a computer program. Fundamentally, it encapsulates the state and behavior of the concept it represents. It encapsulates state through data placeholders called attributes (or member variables or instance variables); it encapsulates behavior through reusable sections of code called methods.

1.6 A simple Java Program

Let's look into our first program. Save the following text into the file called HelloWorld.java

Example Program No 1.1

```
class DearStudent {
    public static void main(String[] args)
    {
```

```
System.out.println ("Java programming much easier with Core Java for U text book.");   }
}
```

Program Explanation

In the above program as it is to compile and execute just to begin with. The main method above is same always in every java program with out any change. Main method using java keywords public and static which will be explained in detail in further chapters. Other things void and String data type having same meaning assuming that you learned in C language.
Remember that almost every statement in Java ends with a semicolon (;).

Compile the above program with this command: javac DearStudent.java.

After compilation successful use the following to run your java program:

Java DearStudent

When you run the program at the command line, you'll see the output:

Java programming much easier with Core Java Hand Book.

1.7 JAVA Variables

A variable refers to the memory location that holds values like: numbers, texts etc, in the computer memory. A variable is a name of location where the data is stored when a program executes.

Java variables can be categorized into the following seven types:

1. Class Variable
2. Instance Variable
3. Array Component Variable
4. Method Parameter Variable
5. Constructor Parameter Variable
6. Exception Handler Parameter Variable
7. Local Variable

1) Class Variable

A java API (Library) class variable is a field declared using the keyword static within a java Class, (or) with or without the keyword static within a java interface declaration.

2) Instance Variable

Java variables that are declared without static keywords are instance variables.

3) Array Component

Array components are unnamed java variables that are created and initialized to default values whenever a new java array object is created.

4) Method Parameter

Java variables declared in the method declaration signature are method parameter variables. Whenever a java method is invoked a variable is created in the same name as it is declared.

5) Constructor Parameter

This is similar to the java method parameter variable. The same way, for all the java variables declared in the constructor variable is created whenever it is invoked.

6) Exception Handler Parameter

Java variables that are declared in the catch clause of a java exception handling mechanism. Whenever a java exception is caught, exception handler parameter variable is created.

7) Local Variable

Java variables that are declared in a block inside a java method (or) for loop is called a java local variable.

1.8 JAVA Keywords

Keywords are reserved words that are predefined in the language; see the table below. All the keywords are in lowercase. These keywords cannot be used as the names for a variable, class or method.

Abstract	boolean	break	byte	case	catch
char	class	const	continue	default	do
double	else	extends	final	finally	float
for	goto	if	implements	import	instanceof
int	interface	long	native	new	package
private	protected	public	return	short	static
strictfp	super	switch	synchronized	this	throw
throws	transient	try	void	volatile	While

Table 1.1 Java Keywords

In addition to the keywords, java reserves the following: true, false, and null. These are values defined in java. You may not use these words for the names of variables, classes, and so on

1.9 Identifiers

Identifiers are used for the class names, method names, and variable names. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters. They must not begin with a number. Java is a case sensitive language.

Some examples:

Valid identifiers : AvgCount abc abc_cde

Invalid identifiers: 2count high-temp

1.10 Operators

Java supports different types of operators which work similar to other programming languages like C language.

1. The Arithmetic Operators
2. The Relational Operators
3. The Bitwise Operators
4. The Logical Operators:
5. The Assignment Operators
6. Conditional Operator (? :)

1.11 Control Statements

The control statements are used to control the flow of execution of the program. This execution order depends on the supplied data values and the conditional logic. Java contains the following types of control statements:

1. Selection Statements: if, if-else, switch.
2. Repetition Statements: while, do-while, for.
3. Branching Statements: break, continue, and return.

The implementation of these is same as C/C++. So it is advised to revise once as C language is prerequisite to this course.

1.12 Data Types

The data types in the Java programming language are divided into two categories and can be explained using the following hierarchy structure

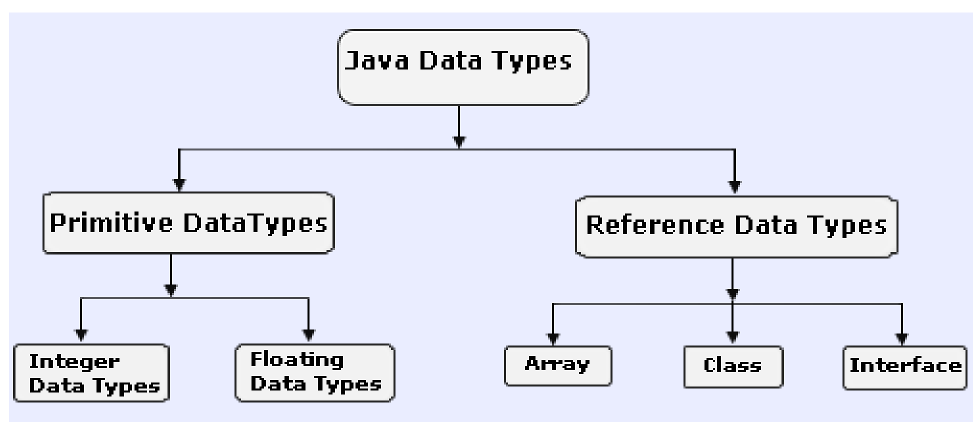


Fig: 1.4 Java Data Types

Below there is a description of the data types and the size allocated in memory when mentioned in the program.

Keyword	**Description**	**Size**
Byte	Byte Length Integer	8 Bit/1 Bytes
Short	Short Integer	16 Bit/2 Bytes
Int	Integer	32 Bit/3 Bytes
Long	Long Integer	64 Bit/4 Bytes
Float	Single Precision Floating point	32 Bit/3 Bytes
Double	Double precision floating point	64 Bit/4 Bytes
Boolean	True/False	True/False
Char	A single character	16 Bit/2 Bytes

Table 1.2 Java Data Types

Unicode

Unicode used 16 bit number to store information about Character & symbols, whereas ASCII used 8 bit. Thus Unicode can represent more than 65000 characters/symbols as compared to ASCII only 256 characters. Unicode characters are standardized to represent most of commonly used languages worldwide including Asian/Indian languages. This is helping in common standard for document generation in any language independent of any custom

software using custom mapping of characters. You can read E-mail message worldwide, read message in browsers in any language.

Type Casting

Casting is the explicit or implicit modification of a variable's type. Remember before casting allows us to view the data within a given variable as a different type than it was given.

In C language casting can be done as follows:

1. Short x = 10;
2. int y = (int)x;

There are two types of conversions that can be done to data in Java Language.

Widening conversions Widening conversions can be done implicitly (by the compiler).

Narrowing conversions Narrowing conversions must be done explicitly, by you, the programmer.

Some examples:

Widening

```
byte x = 100; // Size = 1 byte
// In memory: x =         01100100 = 100
short y;      // Size = 2 bytes
// In memory: y = 0000000000000000 = 0
y = x;        // Legal
// In memory: y = 0000000001100100 = 100
```

Narrowing

```
short x = 1000; // Size = 2 bytes
// In memory: x = 0000001111101000 = 1000
byte y;         // Size = 1 byte
// In memory: y =         00000000 = 0
y = x;          // Illegal
// Not enough space to copy all data
y = (byte)x;    // Legal
// In memory: y =         11101000 = 232
```

Example Program No 1.2

Type Casting refers to changing an entity of one data type into another.

```
Public class conversion {

  public static void main(String[] args)
{
    boolean t = true;  byte b = 2;
    short s = 100;
    char c = ‘C’;
    int i = 200;
    long l = 24000;
   float f = 3.14f;
    double d = 0.000000000000053;
    String g = “string”;
    System.out.println(“Value of all the variables like”);
    System.out.println(“t = “ + t );
    System.out.println(“b = “ + b );
    System.out.println(“s = “ + s );
    System.out.println(“c = “ + c );
    System.out.println(“i = “ + i );
    System.out.println(“l = “ + l );
    System.out.println(“f = “ + f );
    System.out.println(“d = “ + d );
    System.out.println(“g = “ + g );
    System.out.println();
    //Convert from boolean to byte.
    b = (byte)(t?1:0);
    System.out.println(“Value of b after conversion : “ + b);
    //Convert from boolean to short.
    s = (short)(t?1:0);

    System.out.println (“Value of s after conversion : “ + s);
    //Convert from boolean to int.
    i = (int)(t?1:0);

    System.out.println (“Value of i after conversion : “ + i);
    //Convert from boolean to char.
    c = (char)(t?’1’:’0’);

    System.out.println (“Value of c after conversion : “ + c);
    c = (char)(t?1:0);
```

```
System.out.println ("Value of c after conversion in boolean : " + c);
//Convert from boolean to long.
l = (long)(t?1:0);

System.out.println ("Value of l after conversion : " + l);
//Convert from  boolean to float.
f = (float)(t?1:0);

System.out.println ("Value of f after conversion : " + f);
//Convert from boolean to double.
d = (double)(t?1:0);

System.out.println("Value of d after conversion : " + d);
//Convert from boolean to String.
g = String.valueOf(t);

System.out.println("Value of g after conversion : " + g);
g = (String)(t?"1":"0");

System.out.println("Value of g after conversion : " + g);
int sum = (int)(b + i + l + d + f);
System.out.println("Value of sum after conversion : " + sum);
  }
}
```

1.13 JAVA API (Application Programming Interface)

Java API (Application programming interface) is a set of classes and interfaces that comes with the JDK. Java API is actually a huge collection of library routines that performs basic programming tasks such as looping, displaying GUI form etc.

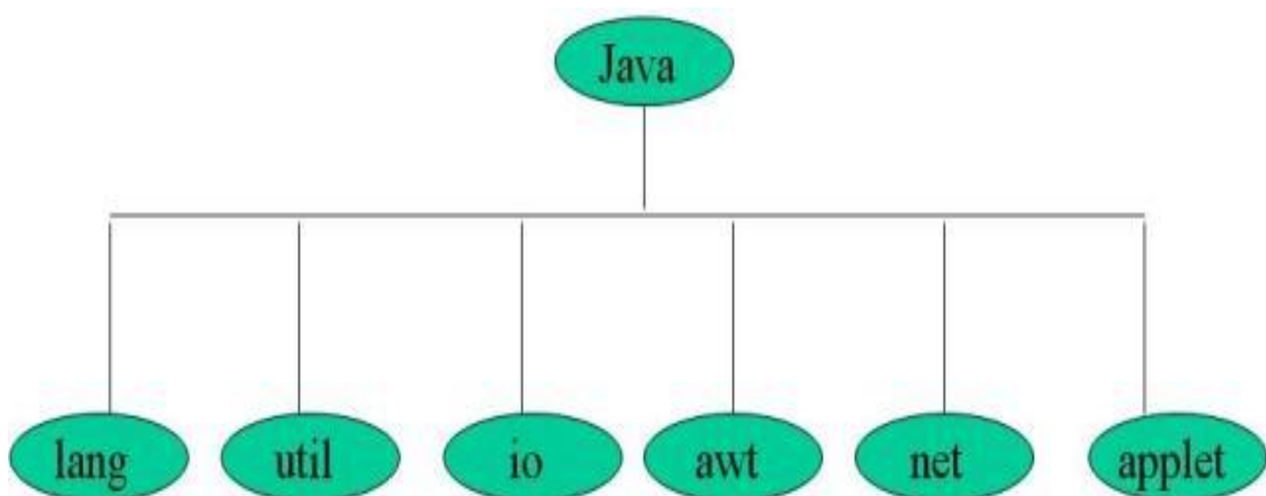


Fig: 1.5 Java API

Java API provides a large number of classes grouped into different packages according to functionality.

java.lang Language support classes. These are the classes that Java compiler itself uses and therefore they are automatically imported. They include classes of primitive types, strings, math functions, threads and exceptions.

java.util Language utility classes such as vector, hash tables, random numbers, data etc.

java.io Input/output support classes. They provide facilities for the input and output of data

java.awt set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on.

java.net Classes for networking. They include classes for communicating with local computers as well as with internet servers.

java. Applet Classes for creating and implementing applets

1.14 Arrays

As we declare a variable in Java, an Array variable is also declared the same way. Array variable has a type and a valid Java identifier i.e. the array's type and the array's name. By type we mean the type of elements contained in an array. To represent the variable as an Array, we use [] notation. These two brackets are used to hold the array of a variable.By array's name, we mean that we can give any name to the array, however it should follow the predefined conventions.

Array declaration

Syntax

```
int[] array_name;           //declares and array of integers. Single dimension
        or
int array_name[];
string[] names;
int[][] matrix;             //this is an array of arrays. Multi Dimension.
```

It is essential to assign memory to an array when we declare it in java. Memory is assigned to set the size of the declared array using new keyword.

Example Program No 1.3

```
public class Array
{
public static void main(String[] args)
```

```
{
Int[] a=new int[5];
}
}
```

Program Explanation

The array ‘a’ will refer to an array of 5 integers.

Array Initialization

After declaring an array variable, memory is allocated to it. The “new” is a java operator is used for the allocation of memory to the array object. The correct way to use the “new” operator is:

```
names = new String[10];
```

Here, the new operator is followed by the type of variable and the number of elements to be allocated.

Example Program No 1.4

```
public class Sum{

public static void main(Sting[] args){
int [] x=new int[101];
for (int i=0;i<x.length;i++)
x[i]=i;
int sum=0;
for(int i=0;i<x.length; i++)
sum+=x[i];
System.out.println(sum);
}
}
```

Program Explanation

In this example, a variable ‘x’ is declared which has a type array of int. that is, int[]. The variable x is initialized to reference a newly created array object. The expression ‘int[]=new int[50]’ specifies that the array should have 50 components.

1.15 The Concept of Classes and Objects

Class

A *class* is a collection of data and methods that operate on that data.

Syntax

```
class identifier
{
data members;
member functions;
}
```

Example

```
Class class_name
{
int a, b;               //data members
public int method_name()            //methods
{
System.out.println("It is a usage of java class example");
}
}
```

1.16 Object

An object is a physical (or) concept representation having pre- defined properties and behavior.Objects are miniature programs that are complete with their own private copy of all internal data.

Java is a heavily object-oriented programming language. When you do work in Java, you use objects to get the job done. You create objects, modify them, move them around, change their variables, call their methods, and combine them with other objects. You develop classes, create objects out of those classes, and use them with other classes and objects.

Declaring Objects

Syntax

<class name> <object name>;

Example

Declaring a Player object

Player player_obj; // declares object

Declaring an object does not create an object. It simply sets up a named location in memory that stores a reference (address) to that object.

To create an object you use the syntax

<object name> = **new** <class name> (<arguments>);

Example

Creating a Player object

Player_ obj = **new** Player(); // creates object

Declaration and creation can be done all in one step. An Object can also be created as follows.

Player player_obj = **new** Player(); // declares and creates object

In Java, you create an object by creating an *instance of a class (*or), in other words, ***instantiating a class.***

An other example is to create Object in the following ways:

Date today; / / today is an object variable of type Date

Object can also be declared by creating with new keyword **new**

Date today = new Date(1996, 8, 30);
(initialize date object and Date is a class in the java.util package).

MyStudent student_obj = **new** MyStudent();

This statement creates an object named student_obj.

Usage of new

The new operator dynamically allocates memory for an object. It has the general form

class_var = **new** classname ();

Here, class-var is a variable of the class type being created. The classname is the name of the class that is being instantiated. The class name followed by parentheses specifies the constructor for the class. A constructor defines what occurs when an object of a class is created.

1.17 Assigning Object Reference Variables

Reference variable is one which holds reference (points to) to an instance (object) of a class.

Example

```
Test t1 = new test ();
 Test t2 = t1;
```

Here t1 and t2 points to same object of Test

You might think that t2 is being assigned a reference to a copy of the object referred to by t1. That is, you might think that t1 and t2 separate and distinct objects. However, this would be wrong. Instead, after this fragment executes, t1 and t2 will both refer to the same object. The assignment of t1 to t2 did not allocate any memory or copy and part of the original object. It simply makes t2 refer to the same object, as does t1. Thus, any changes made to the object through t2 will affect the object to which t1 is referring, since they are the same object.

This situation is depicted here

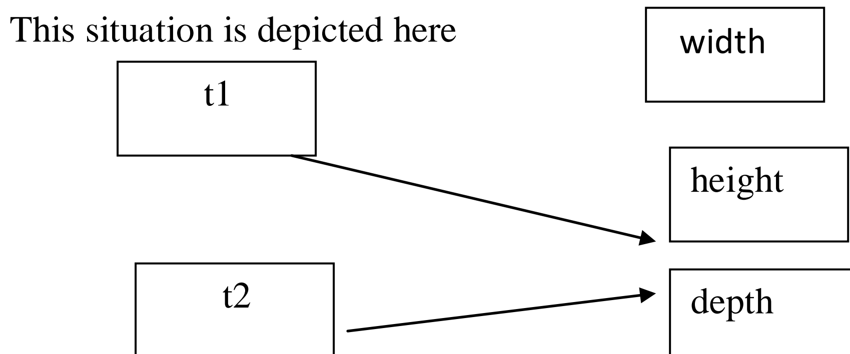


Fig: 1.6: Object reference to the variables

1.18 JAVA Method

A Java method is a collection of statements that are grouped together to perform an operation. A Java method is included inside a Java class. Java methods are similar to functions or procedures in other programming languages. Every Java program must have one main () method to execute.

Syntax

```
modifier returnValueType methodName(list of parameters) {
                                        // Method body;
                                      }
```

Example

```
Public int getStudentDetails (Roll-No String, classno int) {
                // code
return data;
}
```

Example Program No 1.5

```
class Multiple{
int x;
int y;
int z;
int mulmethod(){
return( x * y * z);
}

class DemoofMethods{
public static void main(String args[ ]) {
int result;
Multiple mul1=new Multiple();
mul1.x=10;
mul2.y=20;
mul3.z=30;
result=mul1.mulmethod( );
System.out.println("result is"+res);
}
}
```

Output

Result is 6000.

Program Explanation

The object for the class multiple is declared in the demo of methods class. The values for the method are declared in the main method. The method is called and the result is returned to the main method. We use the term formal parameters to refer to the parameters in the definition of the method. In the example, x and y are the formal parameters. You can remember to call them ``formal'' because they are part of the method's definition, and you can think of a definition as being formal. We use the term actual parameters to refer to variables in the method call, in this case length and width. They are called ``actual'' because they determine the actual values that are sent to the method.

1.19 Constructors

It will be difficult to initialize all the variables in the class each time an instance is created. To minimize this problem java introduced the concept of constructors.Constructors are used to initialize the instance variables (fields) of an object. Constructors are similar to methods, but with some important differences, They are:

1) Constructor name is class name.

2) Default constructor: If you don't define a constructor for a class, a *default Parameter less constructor* is automatically created by the compiler.

3) Constructor's syntax does not include a return type, since constructors neve return a value.

4) When you create a new instance (a new object) of a class using the new keyword , a *constructor* for that class is called. A constructor initializes an object immediately upon creation.

Example Program No 1.6

```
class t2{
t2() {
System.out.println("empty argument called");
}
class t1 {
public static void main(String[] args) {
t2 x1=new t2( );
}
```

Output

Empty argument called

Explanation

When t2 object is created in the class t1, the constructor of that class is called. The output is printed

Difference between method and constructor

A **constructor** is a member function of a class that is used to create objects of that class. It has the same name as the class itself, has no return type, and is invoked using the **new** operator.A method is an ordinary member function of a class. It has its own name, a return type (which may be void), and is invoked using the dot operator.

Parameterized constructor A parameterized constructor in java is same as constructor which take some parameter(s) (variable) ,when it is invoked.

Example Program No 1.7

```
class construct{
int x,y;
construct(int a, int b){
x = a;
```

```
y = b;
}
construct(){
}
int area(){
boolean = x*y;
return(ar);
}
}

public class constructorExam{
public static void main(0String[] args){
construct con = new construct();
con.x = 4;
con.y = 5;
System.out.println("Area of rectangle : " + con.area());
System.out.println("Value of y in Constract class : " + con.y);
construct cons = new construct(2,1);
cons.x = 5;
cons.y = 2;
System.out.println();
System.out.println("Area of rectangle : " + cons.area());
System.out.println("Value of x in Constract class : " + cons.x);
}
}
```

Output

```
Area of rectangle: 20
Value of y in construct class: 5
Area of rectangle: 10
Value of x in the construct class: 5
```

Explanation

The class construct has two constructors, one with parameter-list and other with no parameters. These constructors will be called from the ConstructorExam class, when the object for the construct class is created. The constructor to be called will depend on the type of the object created.

Example Program No 1.8

```
 public class Point {
    int m_x;
    int m_y;

// Constructor
```

```
    public Point(int x, int y) {
        m_x = x;
        m_y = y;
    }

//Parameterless default constructor
public Point() {
        this(0, 0);  // Calls other constructor
}
}
```

Explanation

This keyword: **this** is a reference to the *current object* — the object whose method (or) constructor is being called. In the above program Point is constructor with empty parameter and two parameters, this keyword calling constructor with two parameters as shown above, **this** is used only with in the class.

Super keyword: Using **super** to call a constructor in a parent class. Calling the constructor for the super class must be the *first statement* in the body of a constructor.

1.20 Access Control

Access level modifiers determine whether other classes can use a particular field or invoke a particular method. There are two levels of access control:
The following table shows the access to members permitted by each modifier.

Access Levels				
Modifier	**Class**	**Package**	**Subclass**	**World**
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Table 1.3 Different access modifiers

The first data column indicates whether the class itself has access to the member defined by the access level. As you can see, a class always has access to its own members. The second column indicates whether classes in the same package as the class (regardless of their parentage) have access to the member. The third column indicates whether subclasses of the class — declared outside this package — have access to the member. The fourth column indicates whether all classes have access to the member.

Garbage Collection

The Java virtual machine’s heap stores all objects created by a running Java application. Garbage collection is the process of automatically freeing objects that are no longer referenced by the program. The name “garbage collection” implies that objects no longer needed by the program are “garbage” and can be thrown away. It’s like “memory recycling.”

Usage of static with data and methods

Static variables

When a variable is declared with the static keyword, it is called a “class variable”. A class variable can be accessed directly with the class, without the need to create an instance.Without the “static” keyword, it’s called “instance variable”, and each instance of the class has its own copy of the variable. We can access the other variables using the objects.

Example Program No 1.9

```
class t2 {
   int x = 0; // instance variable
   static int y=0; // class variable
   void setX (int n) { x = n;}
   void setY (int n) { y = n;}
   int getX () { return x;}
   int getY () { return y;}
}

class t1 {

   public static void main(String[] arg) {

      t2 x1 = new t2();
      t2 x2 = new t2();
      x1.setX(9);
      x2.setX(10);
      System.out.println( x1.getX() );
      System.out.println( x2.getX() );
      System.out.println( t2.y );
      t2.y = 7;
      System.out.println( t2.y );
      x1.setY(t2.y+1);
      System.out.println( x1.getY() );
   } }
```

Explanation

We have to observe that the class variable y is called without creating the object. And the instance variable is called with the creation of the object.

1.21 Static Methods

There are two types of methods:
Instance methods are associated with an object and use the instance variables of that object.This is the default. Static methods use no instance variables of any object of the class they are defined in. A static method is called by prefixing it with a *class name*,
e.g., Math.max (i, j);

Example Program No 1.10

```
class static Demo{
static int a=42;
static int b=55;
static void callme(){
System.out.println("a="+a);
}
}
class StaticByName{

        public static void  main(String args[]){

        staticDemo.callme();

        System.out.println("b="+StaticDemo.b);
        }
}
```

Explanation

In the above example the static method callme is called without the creation of objects. The static variable int is also called without the object declaration.

1.22 Usage of Final with Data

- Java variable can be declared using the keyword final. Then the final variable can be assigned only once.

  Example: public class Physics {
  public static final double c = 2.998E8;
  }

- Java classes declared as final cannot be extended. Restricting inheritance.

Example: public final class String

- Methods declared as final cannot be overridden

Example: public final String convertCurrency()

Example Program No 1.11

```
public  final class A{

final Boolea{

System.out.println("this is in method A");
}

//Here we can't write method..We cannot override the method when declared as
final.
System.out.println("this is class A");
}
//class B extends A will lead to error. Does not support inheritance.
Public class FinalExample{

public static void main(String[ ] args){
classA a=new classA();

final int hoursInDay=24;                 //hoursInDay=12; this will not be executed.
System.out.println(" Hours in 5 days"+hoursInDay*5");
```

Output

This is in 31oolea
This is in class A
Hours in 5 days: 120

Explanation

The variable declared as final cannot be declared again. The method declared as final cannot be overridden; The class declared as final cannot be extended.

1.23 Command Line Arguments in Java Program

Java application can accept any number of arguments directly from the command line. The user can enter command-line arguments when invoking the application. When running the java program from java command, the arguments are provided after the name of the class separated by space. For example, suppose a program named CmndLineArguments that accept command line arguments as a string array and echo them on standard output device.

Example Program No 1.12

```
class CmndLineArguments {
 public static void main(String[] args) {
 int length = args.length;
 if (length <= 0) {
 System.out.println("You need to enter some arguments.");
 }
 for (int i = 0; i < length; i++) {
System.out.println(args[i]);
 } } }
```

Output of the program

Run the program with some command line arguments like:
Java cmndLineArguments Mahendra zero one two three

Output
Command Line arguments passed:
Mahendra
Zero
One
Two
Three

1.24 Overloading Constructors

If two constructors of a class (whether both declared in the same class, or both inherited by a class, or one declared and one inherited) have the same name but different signatures, then the method name is said to be overloaded. Methods are overridden on a signature-by-signature basis.Method overloading is one of the ways that Java implements polymorphism.

Example Program No 1.13

```
public class ConstructorOverloading{

  public static void main(String args[]){

    Rectangle rectangle1=new Rectangle(2,4);

    int areaInFirstConstructor=rectangle1.first();

    System.out.println(" The area of a rectangle in

   first constructor is :  " + areaInFirstConstructor);
```

```
    Rectangle rectangle2=new Rectangle(5);

    int areaInSecondConstructor=rectangle2.second();

    System.out.println(" The area of a rectangle in

    first constructor is :  " + areaInSecondConstructor);
    Rectangle rectangle3=new Rectangle(2.0f);

    float areaInThirdConstructor=rectangle3.third();
    System.out.println(" The area of a rectangle in first
        constructor is :  " + areaInThirdConstructor);

    Rectangle rectangle4=new Rectangle(3.0f,2.0f);

    float areaInFourthConstructor=rectangle4.fourth();

    System.out.println(" The area of a rectangle in fourth
      constructor is :  " + areaInFourthConstructor);
  }
}

class Rectangle{
  int l, b;
  float p, q;
  public Rectangle(int x, int y){
    l = x;
    b = y;  }
  public int first(){
    return(l * b);  }
  public Rectangle(int x){
    l = x;
    b = x;  }
  public int second(){
    return(l * b);  }
public Rectangle(float x){
      p = x;
         q = x;  }
  public float third(){
    return(p * q);  }
  public Rectangle(float x, float y){
    p = x;
    q = y;  }
  public float fourth(){
    return(p * q); } }
```

Output

The area of a rectangle in first constructor is : 8

The area of a rectangle in second constructor is : 25
The area of a rectangle in third constructor is : 4
The area of a rectangle in fourth constructor is : 6

Explanation

In the above example there are four constructors named rectangle with difference in their parameter list. As you see, the proper overloaded constructor is called based upon the parameters specified when 'new' is executed.

1.25 Overloading Methods

Method Overloading is similar to constructor overloading. Here we overload methods. Method overloading is one of the ways java implement polymorphism.

Example Program No 1.14

```
Class Life {
String style;
String dressing;
int age;
public int westernliving(int a,String b){
}
public void westernliving() {
}
public String westernliving(String c) {  }
```

Explanation

In the above example when the method is invoked, it checks for the same argument list. If they are same then the method will be called.

Example Program No 1.15

```
class OverloadDemo {
void test() {
System.out.println("No parameters");
}
// Overload test for one integer parameter.
void test(int a) {
System.out.println("a: " + a);
}
// Overload test for two integer parameters.
void test(int a, int b) {
```

```
System.out.println("a and b: " + a + " " + b);
}
// overload test for a double parameter
double test(double a) {
System.out.println("double a: " + a);
return a*a;
}
}
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
double result;
// call all versions of test()
ob.test();
ob.test(10);
ob.test(10, 20);
result = ob.test(123.2);
System.out.println("Result of ob.test(123.2): " + result);
}
}
```

Output

```
No parameters
a: 10
a and b: 10 20
double a: 123.2
Result of ob.test (123.2): 15178.24
```

Explanation

As you can see, test( ) is overloaded four times. The first version takes no parameters, the second takes one integer parameter, the third takes two integer parameters, and the fourth takes one double parameter. The fact that the fourth version of test( ) also returns a value is of no consequence relative to overloading, since return types do not play a role in overload resolution.When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters.

1.26 Parameter Passing – Call by Value

All the primitive or the simple data types (int,float,35oolean etc) are passed as call by value whereas the abstract data types(class objects) are by call by reference.The difference is that the call by value method copies the value of an

argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument. The second way an argument(not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that the changes made to the parameter will affect the argument used to call the subroutine.

Example Program No 1.16

```
class Demo{
void add(int a, int b) {
a=a+2;
b=b*2;
}
}
class callByValue{
public static void main(String args[]){
Demo de=new Demo();
int i=5; j=6;
System.out.println("The value of I and j before calling"+a "and"+b);
De.add(i,j);
System.out.println("the value of I and j after call"+a "and"+b);
}
```

Output

The value of I and j before calling: 5, 6
The value of I and j after calling: 5,6

Explanation

The values of I and j does change before and after the method add, because the value passed is just the copy of the value but not the original. So the changes made in the method will not change the actual value.

Example Program No 1.17

```
class Demo{
void add(int a, int b) {
a=a+2;
b=b*2;
}
}
class callByRef{
public static void main(String args[]){
Demo de=new Demo();
int i=5; j=6;
```

```
System.out.println("The value of I and j before calling"+a "and"+b);
De.add(i,j);
System.out.println("the value of I and j after call"+a "and"+b);
}
}
```

Output

The value of I and j before calling: 5, 6
The value of I and j after calling: 7, 12

Explanation

The values of I and j change before and after the method add, because the value passed is reference of the value . So the changes made in the method will change the actual value.

1.27 Nested and Inner Classes

A nested class is defined inside another class. There are two types of nested classes: static nested classes and inner classes. A static nested class is declared inside another class with the static keyword or within a static context of that class. A static class has no access to instance-specific data. An inner class is a nonstatic class declared inside another class. It has access to all of its enclosing class's instance data, including private fields and methods.

Example Program No 1.18

```
public class Main {
public static void main(String[] args) {
 outterclass outobj=new outterclass();
  outterclass.innerclass  innerobj=outobj.new  innerclass();
    }}
class outterclass {
int outer_x=100;
    class innerclass
    {}}
```

Explanation

In the above example the variables initialized in the outer class can be accessed in the inner class. It means the inner class methods are declared under the scope of outer class.

1.28 Exploring the String Class

Strings, which are widely used in Java programming, are a sequence of characters. In the Java programming language, strings are objects.

The String class implements immutable character strings, which are read-only once the string object has been created and initialized. Objects of type String are read only and immutable. The StringBuffer class is used to represent characters that can be modified.All string literals in Java programs are implemented as instances of String class.

Examples

1. String str1 = "I can't be changed once created!"

A Java string literal is a reference to a String object **.**Some useful methods for String objects are equals( ) and substring( ). The equals( ) method is used for testing whether two Strings contain the same value. The substring ( ) method is usedobtain a selected portion of a String. The length() method is used to find the length of the given string.

1. int myLenght = "Hello world".length();
2. String finalString = "Hello" + "World";
3. String str1="Hello"; String str2="Hello";
   If(str1 == str2)   System.out.println("Equal");

As with any other object, you can create String objects by using the new keyword and a constructor.

1. String finalString = new StringBuffer().append("Hello").append("World").toString();

SUMMARY

1. Java is an object oriented programming language.
2. Object Oriented Programming techniques include features such as data abstraction, encapsulation, modularity, polymorphism, and inheritance
3. The size of the java datatypes is not the same as c. Since java uses Unicode.
4. Class: A Java class is a group of java methods and variables.
5. The syntax of the array in java is different from array in c. The syntax is
   data_type[] arrayname=new datatype[size];
6. A Java object is a set of data combined with methods for manipulating that data. An object is made from a class; a class is the blueprint for the object.
7. New keyword: The new operator dynamically allocates memory for an object.
8. A Java method is a collection of statements that are grouped together to perform an operation which is similar to functions in C language.
9. Constructors are used to initialize the instance variables (fields) of an object. A java constructor has the same name as the name of the class to which it belongs. Constructor's syntax does not include a return type

10. This and super keywords in constructor: this in constructor will call another constructor of the class. Super will call the constructor in the super class.
11. Constructor overloading: When there is more than one constructor of the class with the same name but the difference in the parameter-list, it is known as constructor overloading.
12. The keyword final can be declared before the variable, method and a class.
13. Access level modifiers determine whether other classes can use a particular field or invoke a particular method. The types of modifiers present are public, private, and no modifier are protected.
14. Java application can accept any number of arguments directly from the command line. These are known as command line arguments.
15. In Java we can declare a class inside another class. This is known as nested class.
16. The Java String class implements immutable character strings, which are read-only once the string has been created and initialize.

****

CHAPTER-2

INHERITANCE, PACKAGES, INTERFACES AND I/O STREAMS

2.1 Inheritance

It is one of the most important features of Object Oriented Programming. It is the concept that is used for reusability purpose. Inheritance is the mechanism through which we can derive classes from other classes. The derived class is called as child class or the subclass or extended class and the class from which we are deriving the subclass is called the base class or the parent class.

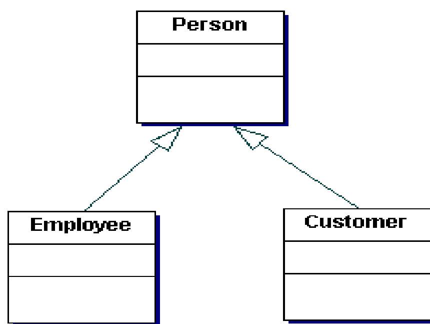
Definition

Inheritance can be defined as the process where one object acquires the properties of another.Inheritance is a major component of object-oriented programming. Inheritance will allow you to define a very general class, and then later define more specialized classes by simply adding some new details to the older more general class definition. To derive a class in java the keyword 'extends' is used.

IS-A Relationship IS-A is a way of saying. This object is a type of that object. Let us see how the extends keyword is used to achieve inheritance. Inheritance describes an IS-A relationship. Dalmatian IS-A Dog.

HAS-A relationship This determines whether a certain class HAS-A certain thing. A Body HAS-A Leg. HAS-A relationships are implemented with embedded references to other objects
Java only supports only single inheritance. This means that a class cannot extend more than one class.

class person extend employee class person
implements employee



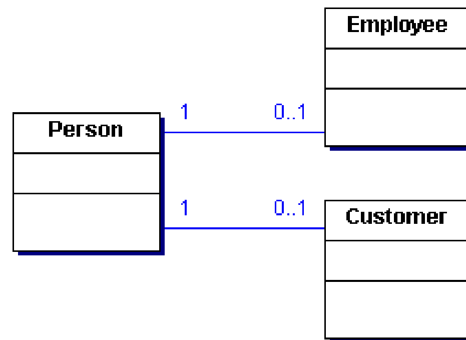


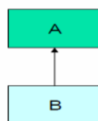
Fig2.1 Inheritance

To inherit a class, you simply incorporate the definition of one class into another by using the extends keyword.

2.1.1 Forms of Inheritance

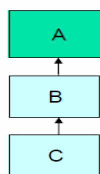
Java supports three forms of inheritance as below:

1. **Single level Inheritance**



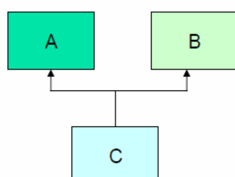
In this type of inheritance "class B extends A".

2. Multi Level Inheritance



In this type of inheritance "class C extends class B and class B extends class A.

3. Multiple Inheritance



In this type of inheritance "class c inherits features of class A as well the features of class B". This can be implemented through interfaces.

Examples For Types Of Inheritance

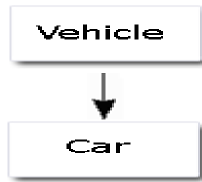


Fig 2.1: Single Inheritance **Fig 2.2: Multi Level Inheritance**

Defining A Sub Class

A subclass/child class is defined as follows:

```
class SubClassName extends SuperClassName
{
fields declaration;
methods declaration;  }
```

Example Program No 2.1

```
public class MyBase{
public int x;
public void show(){
System.out.println(“x=”+x);
}
class MyDerived extends MyBase{
public int y;
public void show() {
System.out.println(“x=’+x);
System.out.println(“y=”+y);
}}
```

Explanation

In the above example we can directly invoke the value of x of the base class from the derived class.

Member access

Though we can access all the members of the super class using inheritance we can restrict it by using private to the member variable.

2.1.2 Usage of super key word

The super is java keyword. As the name suggest super is used to access the members of the super class. It is used for two purposes in java.

1. The first use of keyword super is to access the hidden data variables of the super class hidden by the sub class. This functionality can be achieved just by using the following command.:**super.member;**

2. The second use of the keyword super in java is to call super class constructor in the subclass. This functionality can be achieved just by using the following command.**super(param-list);**

Example Program No 2.2

```
class A{
  int a;
 float b;
  void Show(){
   System.out.println("b in super class:  " + b);
  }}
class B extends A{
  int a;
  float b;
  B( int p, float q){
   a = p;
   super.b = q;
  }
  void Show(){
   super.Show();
   System.out.println("b in super class:  " + super.b);
   System.out.println("a in sub class:    " + a);
}
  public static void main(String[] args){
   B suburb = new B(1, 5);
   subobj.Show();
  }
}
```

Output

B in super class: 5
B in subclass:1

Explanation

Although the instance variable a in **B** hides the a in **A**, **super** allows access to the a defined in the super class.

Second usage of super keyword

Example Program No 2.3

```
class A {
A()
 {
 System.out.println("A class Constructor");  }}
class B extends A {
B() {
super(); //invoke superclass constructor
}
public static void main(String a[]) {
B obj=new B(); //At the initiation time itself, it invoking B() and A()
} }
```

Output

A class constructor

Explanation

When the object is created in the class B, At the initiation time itself, it invokes B() and A().

2.2 Method Overriding

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

When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. A method is said to be overridden when one is in parent class and another is in child class with the same name, same return type, and same parameter. When you extend a class, you can change the behavior of a method in the parent class. Overriding allows a subclass to **re-define** a method it inherits from its superclass.

When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. A method is said to be overridden when one is in parent class and another is in child class with the same name, same return type, and same parameter.

Example Program No 2.4

```
class A {
int i;
A(int a, int b) {
i = a+b;  }
void add() {
System.out.println("Sum of a and b is: " + i);
} }
class B extends A {
int j;
B(int a, int b, int c) {
super(a, b);
j = a+b+c;  }
void add() {
super.add();
System.out.println("Sum of a, b and c is: " + j);
}}
class MethodOverriding {
public static void main(String args[]) {
B b = new B(10, 20, 30);
b.add(); } }
```

Output

Sum of A and B is 30
Sum of A, B and C is 60

Explanation

When add() is invoked on an object of type B, the version of add() defined within B is used. That is, the version of add() inside B overrides the version declared in A. If you want to access the superclass version of the overridden function then we must use the super keyword. This will invoke the add() in the superclass.

2.3 Abstract Class

An abstract class is a class that is declared by using the abstract keyword. It may or may not have abstract methods. Abstract classes cannot be instantiated, but they can be extended into sub-classes. The key idea with an abstract class is useful when there is common functionality that's like to implement in a superclass and some behavior is unique to specific classes. The methods without the abstract keyword are called as concrete methods.

Sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. In that case we will use the abstract classes. The super class just defines the method and the subclass implements those methods.

Must Remember

1) Abstract class contains abstract methods.
2) A Program can't instantiate an abstract class.
3) Abstract classes contain mixture of non-abstract and abstract methods.
4) If any class contains abstract methods then it must implements all the abstract methods

Example Program No 2.5

```
abstract class A {
abstract void callme(); // concrete methods are still allowed in abstract classes
void callmetoo() {
System.out.println("This is a concrete method."); }
}
class B extends A {
void callme() {
System.out.println("B's implementation of callme.");
} }
class AbstractDemo {
public static void main(String args[]) {
B b = new B();
b.callme();
b.callmetoo(); }   }
```

Explanation

No objects of class A are declared in the program. Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because java's approach to runtime polymorphism is implemented through the use of superclass references. Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass.

Can Abstract Class have constructors?

Abstract class's can have a constructor, but you cannot access it through the object, since you cannot instantiate abstract class. To access the constructor create a sub class and extend the abstract class which is having the constructor.

Example Program No 2.6

```
public abstract class AbstractExample {
public AbstractExample(){
System.out.println("In AbstractExample()");
}}

public class Test extends AbstractExample{
public static void main(String args[]){
Test obj=new Test();
}}
```

Explanation

When an object is created for the class Test, it creates an object for the abstract class also, since it is extended. When the object is created, the constructor of the abstract class is called.

2.4 Dynamic Method Dispatch

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

An important principle is a super class reference variable can refer to a subclass object**.**

Example Program No 2.7

```
class A {
void callme() {
 System.out.println("Inside A's callme method");   }}
class B extends A {
  void callme() { System.out.println("Inside B's callme method");  }}
class C extends A {
 void callme() {
System.out.println("Inside C's callme method");  } }
class Dispatch {

  public static void main(String args[]) {

  A a = new A(); // object of type A
  B b = new B(); // object of type B
  C c = new C(); // object of type C
  A r; // obtain a reference of type A
   r = a; // r refers to an A object
  r.callme();     // calls A's version of callme
  r = b;          // r refers to a B object
  r.callme();  // calls B's version of callme
  r = c;         // r refers to a C object
  r.callme();  // calls C's version of callme
} }
```

Explanation

This program creates one superclass called A and two subclasses of it called B and C. Subclasses B and C override callme() declared in A. Inside the main() method, objects of type A,B, C are declared. Also, a reference of type A, called r is declared. The program then assigns a reference to each type of object to r and uses that reference to invoke callme (). The callme () is executed is determined by the type of object being referred to at the time of the call.

2.5 Using Final with Inheritance

- A final class cannot be sub classed
- All methods in a final class are implicitly final.
  Example: public final class MyFinalClass {...}
- A final method cannot be overridden by subclasses
  Example: public class MyClass {
  public final void myFinalMethod() {...}

```
}
```

A final variable can only be assigned once. You can declare a variable in any scope to be final. The value of a final variable cannot change after it has been initialized. Such variables are similar to constants in other programming languages. To declare a final variable, use the final keyword in the variable declaration before the type:

Example: final int aFinalVar = 0;

A compile-time error occurs if the name of a final class appears in the extends clause of another class declaration; this implies that a final class cannot have any subclasses. A compile-time error occurs if a class is declared both final and abstract.

Example Program No 2.8

```
public class A {
 public static final void f() {
System.out.println("A's f()");  }
} // our Class
public class B {
 public void g() {
A.f();  }}
```

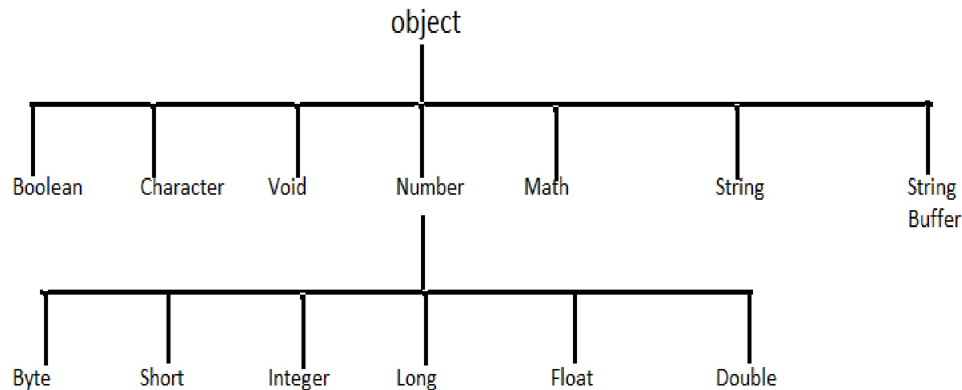
Explanation

If we override the method ‘ f ’ of the class A in the subclass B, we will get the compile time errors.

2.6 Object Class

The Object class sits at the top of the class hierarchy tree in the Java development environment. All other classes are subclasses of the Object.Every class in the Java system is a descendent (direct or indirect) of the Object class. This class is in the package of java.lang which is the default package and is automatically imported in every java program without explicitly is importing it**.** Every class has Object as a superclass. It is in java.lang package referred as java.lang.Object. All classes in Java technology are directly or indirectly derived from the Object class. Some of the subclasses of Object class are - Boolean, Number, Void, Math, String, StringBuffer.

Object Class and some of its sub classes shown as sample to understand



Some Methods of java class (java.lang.Object) in java.lang. Package

Method	Purpose
clone ()	Creates a new object of the same class as this object.
equals (Object)	Compares two Objects for equality.
finalize()	Called by the garbage collector on an object when garbage
	Collection determines that there are no more reference to the object.
getClass( )	Returns the runtime class of an object.
hashCode()	Returns a hash code value for the object.
toString()	Returns a string representation of the object

2.7 Packages

Package = directory.

Definition A Java Package can be thought of as a collection of Java source files, usually grouped by functionality. Java classes can be grouped together in *packages*. A package name is the same as the directory (folder) name which contains the .java files.

You can define the class of the same name in different packages so the programmers will not conflict with each other.

Accessing the packages You can access the class of the other package with the help of access modifiers like public, private.

You declare packages when you define your Java program, and you name the packages you want to use from other libraries in an *import* statement.
The first statement, other than comments, in a Java source file, must be the package declaration.

Creating Packages

Creating a Java Package is relatively simple but may seem a bit confusing for people who have not done it before. There are two fundamental requirements for a package to exist:

- The package must contain one or more classes or interfaces. This implies that a package cannot be empty.
- The classes or interfaces that are in the package must have their source files in the same directory structure as the name of the package.

Example of package

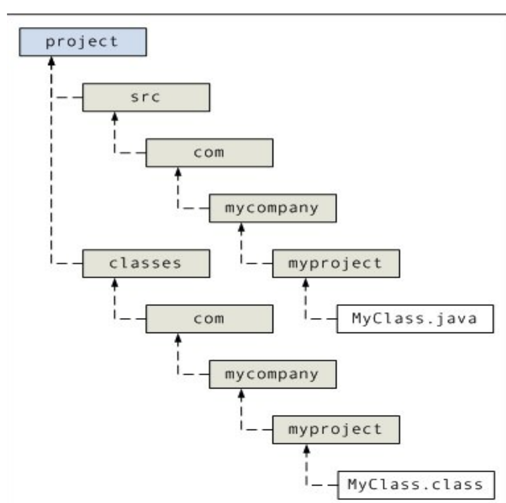
Package Students; class MyClass { //Code…. }

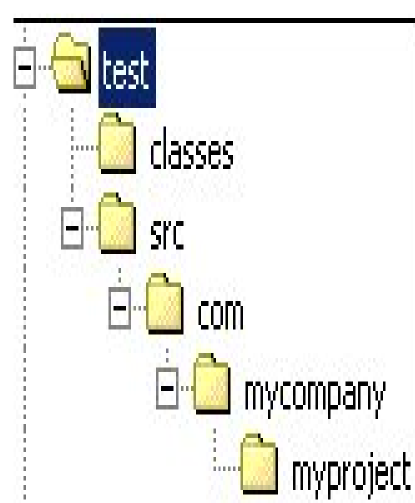
Importing a Single Package Member

```
import java.util.Vector;
class Test {
 Vector vector;
Test() {
vector = new Vector();
 } }
```

Importing an Entire Package

```
 import java.util.*;    class Test { ... }
```





MyClass in the package named com.mycompany.myproject and all source code in the src directory. This can be written in java program

package com.mycompany.myproject;   public class MyClass { }

2.7.1 Scope of the variables

The purpose of access specifiers is to declare which entity cannot be accessed from where. Its effect has different consequences when used on a class, class member (variable or method), and constructor.

Access Specifiers for Class Variables and Methods

Below is a table showing the effects of access specifiers for **class members**.
"Y" means yes. Blank means No

Specifier	**Class**	**subclass**	**package**	**World**
Private	YES	NO	NO	NO
Protected	YES	YES	YES	NO
Public	YES	YES	YES	YES
(none)	YES	NO	YES	NO

For example, if a variable is declared protected, then the class itself can access it, its subclass can access it, and any class in the same package can also access it, but otherwise a class cannot access it.

If a class member doesn't have any access specifier (the "none" row in above), its access level is sometimes known as "package".

Example Program No 2.9

```
class P {
   int x = 7;
}
public class AS {
   public static void main(String[] args) {
      P p = new P();
      System.out.println (p.x);  }}
```

Explanation

The code compiles and runs. But, if you add “private” in front of “int x”, then you'll get a compiler error: “x has private access in P”. This is because when a member variable is private, it can only be accessed within that class

2.8 Interfaces

As we have already discussed that the multiple inheritance can be implemented through interfaces. i.e.; we can inherit the features of two classes to a single class.

Definition An interface is a named collection of method definitions (without implementations). An interface can also include constant declarations.

Implements The keyword **implement** specifies that the given class implements an interface. Implements is a keyword used on class or interface statements. When a class provides the methods (possibly abstract) necessary to confirm to some interface we say that class implements the interface.

Extending means adding new method definitions. Implementing means satisfying the existing interface contract by writing the prescribed method bodies. Logically, when you extend an abstract class and fill in the method bodies, that should be called implementing too, but in the Java world that is called extending.

An interface defines a protocol of behavior. An interface declaration introduces a new reference type whose members are classes, interfaces, constants and abstract methods. A variable whose declared type is an interface type may have as its value a reference to any instance of a class which implements the specified interface.

It is not permitted to declare an interface as final; it will cause a compilation error. This is a Java language design decision; interface types are intended to be implemented and can be extended without restriction.

Declaring an Interface

public interface <interfacename>

{ returnType methodName1(arguments); }

To access an interface-implementing class with an interface class reference

Syntax [modifiers] InterfaceName variableName;

Example Program No 2.10

```
public class Main {
public static void main(String[] args) {
  shape circleshape=new circle();
     circleshape.Draw();
    }}
 interface shape {
    public   String baseclass="shape";
     public void Draw();
     }
 class circle implements shape {
   public void Draw() {
   System.out.println("Drawing Circle here");
   } }
```

Explanation

The method draw() is declared in the shape interface but the implementation of the draw() method is done in the circle class. We have to define the method declared in implemented interface

When should you use an abstract class, when an interface, when both?

Abstract class is a class which contains one or more abstract methods, which has to be implemented by sub classes. An abstract class can contain no abstract methods also i.e. abstract class may contain concrete methods. A Java Interface can contain only method declarations and public static final constants and doesn't contain their implementation. The classes which implement the Interface must provide the method definition for all the methods present. Abstract classes are useful in a situation when some general methods should be implemented and specialization behavior should be implemented by subclasses. Interfaces are useful in a situation when all its properties need to be implemented by subclasses.

Difference between classes and Interfaces

Sno	Class	Interfaces
1.	classes are used only when there is a “is-a” type of relationship between the classes (Inheritance	Interfaces can be implemented by classes that are not related to one another.
2.	we cannot extend more than one class	You can and implement more than one interface. So multiple inheritance is achieved.
3.	A class can implement any no. of methods	Interfaces cannot implement methods and are abstract
4.	The access specifier public, private, protected with classes	The interface uses only one specifier is public
5.	We can create instances or objects for classes	We cannot create objects or instances for interfaces

Important Points to note

1. There is no difference between a fully abstract class (all methods declared as abstract and all fields are public static final) and an interface.
2. Neither Abstract classes nor Interface can be instantiated. However, if we create an instance of a class that extends an Abstract class, compiler will initialize both the classes.
3. In an abstract class, it is possible that some methods are not implemented. In the case of an interface, none of the methods are implemented. If you would be able to create an instance, and call a non-implemented method, what code should be executed? Not that of the class / interface itself, since there is no code to execute.
4. In Java an interface cannot extend an abstract class. An interface may only extend a super-interface.

2.8.1 Extending Interfaces

Interfaces can have subinterfaces, just as classes can have subclasses. A subinterface inherits all the abstract methods and constants of its superinterface and can define new abstract methods and constants. Interfaces are different from classes in one very important way, however: an interface can have an extend

clause that lists more than one superinterface. For example, here are some interfaces that extend other interfaces:

Example Program No 2.11

```
public interface Sports{
public method1();
}
public interface Football extends Sports{
  public method2();
public method3();
}
class Hockey implements Football{
public void method1(){
System.out.println("this is method 1");
}
public void method2(){
System.out.println("this is method 2");
}
public void method3(){
System.out.println("this is method 3");
}}
public class Mainclass{
public static void main(string args[])
Hockey hoc= new Hockey();
hoc.method1();
hoc.method2();
hoc.method3();
}
```

Explanation

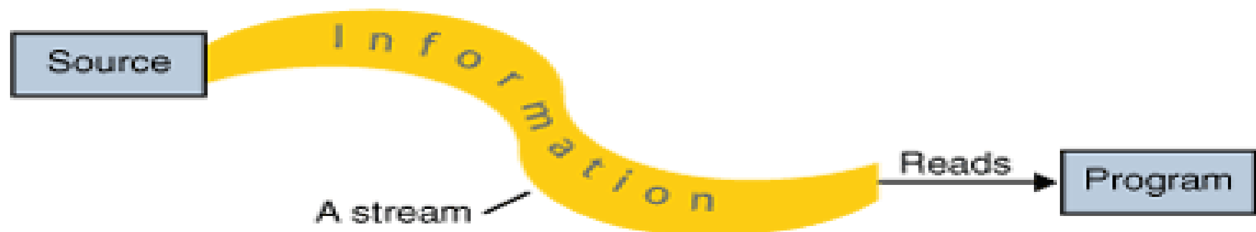
The interface football has two methods, but it inherits the method1 from the interface sports. So the class hockey which implements the interface football needs to define all the three methods i.e., method1, method2, method3.

2.9 I/O Streams

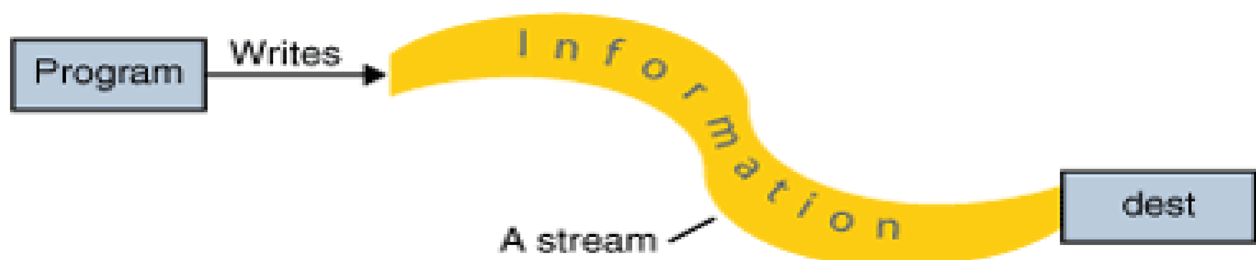
A stream is a sequence of data. An I/O Stream represents an input source or an output destination. A stream can represent many different kinds of sources and destinations. They are: disk files, devices, other programs, a network socket, and memory arrays.Streams support many different kinds of data. They are: simple bytes, primitive data types, localized characters, and objects. Some

streams simply pass on data; others manipulate and transform the data in useful ways.

Reading information into a program

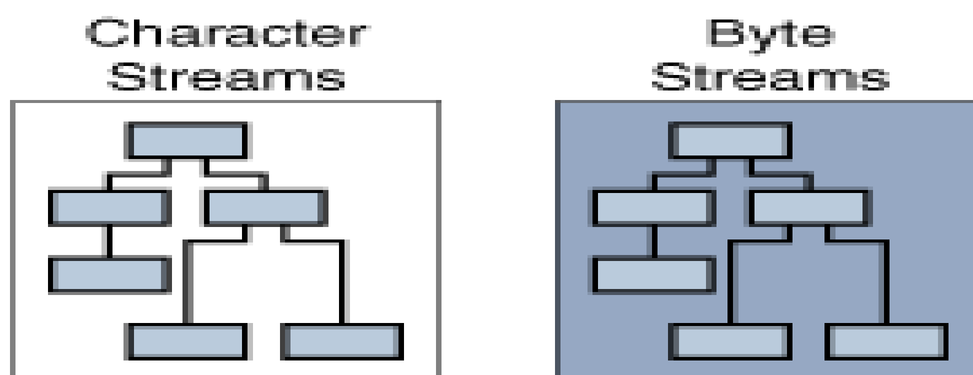


Writing information out of a program



2.9.1 Stream Classes

The java.io package contains a collection of stream classes. To use these classes, a program needs to import the java.io package. The stream classes are divided into two class hierarchies, based on the data type: 1) characters stream 2) bytes stream. Shown in the following figure:

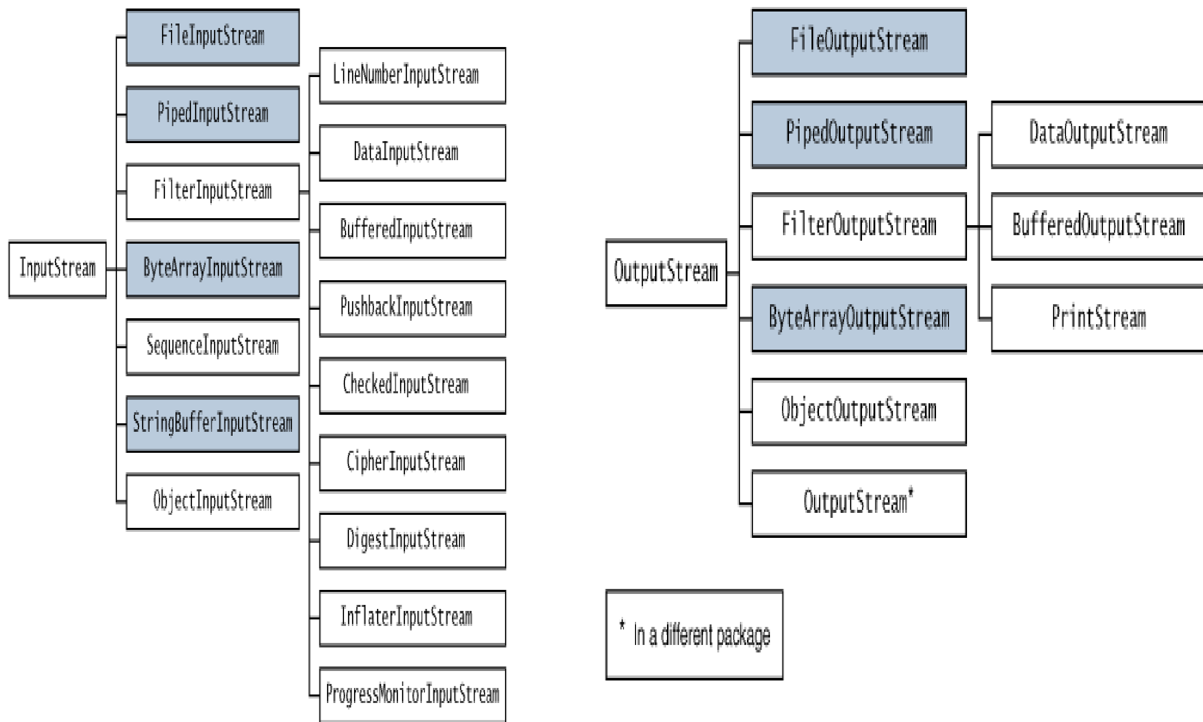


The java.io package contains two independent hierarchies of classes:

1) One for reading and writing bytes
2) The other for reading and writing characters.

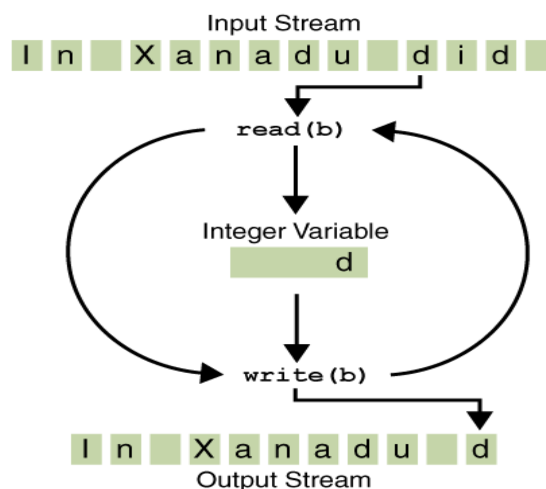
Character Stream classes in java.io.package

To read and write (8-bit) bytes, program should use the byte streams.
InputStream (read bytes) and OutputStream (write bytes) provide the java API



BYTE STREAMS

Programs use byte streams to perform input and output of 8-bit bytes. All byte stream classes are descended from InputStream and OutputStream. A simple loop that reads the input stream and writes the output stream, one byte at a time Simple byte stream input and output below shown:



CHARACTER STREAMS

Character streams is that they make it easy to write programs that are not dependent upon a specific character encoding, and are therefore easy to internationalize. Java stores strings in Unicode, an international standard character encoding that is capable of representing most of the world's written languages. Character streams are that they are potentially much more efficient than byte streams. The implementations of many of Java's original byte streams are oriented around byte-at-a-time read and write operations.

Java Stream Predefined classes

When the JVM is executing a Java application program, it needs a console window to provide 3 predefined input and output data streams to the Java program to:

1) System.in: Standard input stream connected to the console window.
2) System.out: Standard output stream connected to the console window.
3) System.err: Standard error stream connected to the console window.

When you use J2SDK package to execute a Java program, the console window is the same window where you entered the "java" command.

Usage of predefined streams classes example

Example Program No 2.12

```
import java.io.*;
public class Reverser {
public static void main(String[] arg) {
InputStream in1 = System.in;
InputStreamReader in2 = new InputStreamReader(in1);
BufferedReader in = new BufferedReader(in2);
PrintStream out = System.out;
PrintStream err = System.err;
err.println("This program reverses the character positions of ");
err.println("each line from the standard input stream.");
err.println("Enter '.' to end the program.");
try {
 String s = in.readLine();
while (s!=null) {
 char[] a = s.toCharArray();
 if (s.equals("."))
  break;
int n = a.length;
```

```
for (int i=0; i<n/2; i++) {
char c = a[i];
 a[i] = a[n-1-i];
a[n-i-1] = c;
}
out.println(new String(a));
 s = in.readLine(); }
} catch (IOException e) {
err.println(e.toString());
} } }
```

Important note about above example program

The "System.in" is an "InputStream" object, which only offers methods to read in information as bytes.

To be able to read as one "String" per line, I need to convert "System.in" to a "BufferedReader" object. The "while" loop should end when "System.in" reaches the end of the stream. However, HotSpot for Windows gives you no way to indicate the end of the standard input stream from the key board. So I used a special pattern, one and only one character '.' in a line, to end the loop
The 3 lines in the above program are from the "System.err" stream. The 4th, 6th, and 8th lines are copies of the text lines of the "System.in" stream, because the console window is echoing everything I typed in on the keyboard. The 5th and 7th lines are from the "System.out" stream.
Windows system can also redirect the console input and output to files. First, I saved the 2 lines of text and the '.' into a file called input.txt, then run the following command:
c:\jdk\bin\java -cp . Reverser < input.txt > output.txt

Example for using streams to Read and write data to and from files

The following program is a command-line filter uses the pre-defined streams **System.in** and **System.out** to copy standard input to standard output.

Example Program No 2.13

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

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

```
{
 int b;
 while ((b = System.in.read()) != -1)
System.out.write(b);
}
 }
```

Example program No 2.14

```
import java.io.*;
class CopyFile {
public static void main(String[] args) throws IOException {

FileInputStream fin = new FileInputStream( args[0]);

FileOutputStream fout = new FileOutputStream( args[1]);
 int b;
 while ((b = fin.read()) != -1) {
fout.write(b);
fin.close();
fout.close();
 }
```

SUMMARY

1. Inheritance can be defined as the process where one object acquires the properties of another.
2. Java supports three forms of inheritance: Single level Inheritance, Multi level inheritance, multiple inheritances.
3. super keyword is used to access the members of the super class. It is used for two purposes in java. The first use is to access the hidden data variables of the super class hidden by the subclass. Command: **super.member;** the second use is to call super class constructor in the subclass. Command:**super(param-list);**
4. Method Overriding: when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass
5. Abstract class: An abstract class may or may not have abstract methods. Abstract classes cannot be instantiated, but they can be extended into sub-classes. The key idea with an abstract class is useful when there is common functionality that's like to implement in a superclass and some behavior is unique to specific classes.
6. Dynamic method dispatch:It is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
7. Use of final in java: A final class cannot be sub classed.All methods in a final class are implicitly final. A final method cannot be overridden by subclasses. A final variable can only be assigned once.
8. The super class of all the classes in java is object. The Object class sits at the top of the class hierarchy tree in the Java development environment.
9. Interface: An interface is a named collection of method definitions (without implementations). In java, multiple inheritance can be implemented through interfaces.
10. A stream is a sequence of data. An I/O Stream represents an input source or an output destination.
11. The stream classes are divided into two class hierarchies, based on the data type: characters stream and bytes stream.
12. To write a java program there are 3 predefined input and output data streams
    1) System.in: Standard input stream connected to the console window.
    2) System.out: Standard output stream connected to the console window.
    3) System.err: Standard error stream connected to the console window.

****

CHAPTER-3

EXCEPTION HANDLING AND MULTITHREADING

3.1 Fundamentals of Exception Handling

Due to design errors or coding errors, our programs may fail in unexpected ways during execution. It is our responsibility to produce quality code that does not fail unexpectedly.Consequently we must design error handling into our programs.

Definition of Exception

An exception is a problem that arises during the execution of a program.
An exception can occur for many different reasons, including the following:

1) A user has entered invalid data.
2) A file that needs to be opened cannot be found.
3) A network connection has been lost in the middle of communications or the JVM has run out of memory.

3.1.1 Types of Exceptions

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.
The Exception class has two main subclasses: IOException class and RuntimeException Class.

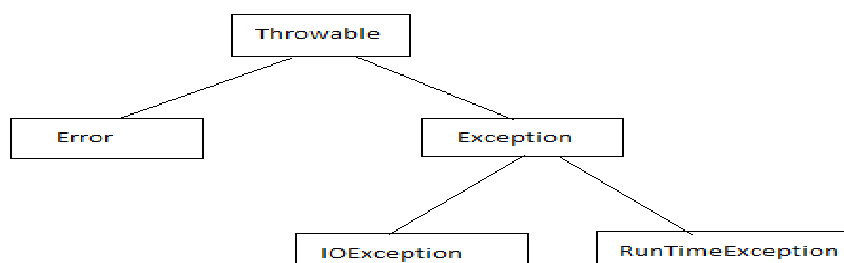


Fig: 3.1 Types of Exceptions

In Java, there are three categories of exceptions

1. **Checked exceptions** A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer

    Example: file cannot be found, an exception occurs

2. **Runtime exceptions (Unchecked Exceptions)** A runtime exception is an exception that occurs that probably could have been avoided by the programmer. Runtime exceptions are ignored at the time of compilation.

    Example: illegal cast operation

3. **Errors** These are not exceptions at all, but problems that arise beyond the control of the user or the programmer.

    Example If a stack overflow occurs, an error will arise

3.2 Usage of Try and Catch

During the program execution if any error occurs and you want to print your own message or the system message about the error then you write the part of the program which generate the error in the try{} block and catch the errors using catch() block. Exception turns the direction of normal flow of the program control and send to the related catch() block. Error that occurs during the program execution generate a specific object which has the information about the errors occurred in the program. When an exception occurs, the code after the exception has occurred will not be executed. The catch statement will be executed.
The try /catch statement is used to handle errors and exceptions that might occur in the code.

The general syntax of the try/catch statement

try {
body-code
}
Catch (*exception-classname variable-name*) {

handler-code
}

The body-code contains code that might throw the exception that we want to handle exception. Classname is the class name of the exception we want to handle. Variable name specifies a name for a variable that will hold the exception object if the exception occurs.handler-code contains the code to execute if the exception occurs.

Example Program No 3.1

```
public class ExcepTest{
 public static void main(String args[]){
   try {
     int a[] = new int[2];
    System.out.println("Access element three :" + a[3]);
      } catch(ArrayIndexOutOfBoundsException e) {
       System.out.println("Exception thrown  :" + e);
      }
    System.out.println("Out of the block");   }
}
```

Output

Out of block

Explanation

The try block is executed and when the array tries to access the element out of the bound, and exception is generated. Then the exception is thrown and the catch block catches that exception. The statement inside the catch block will be printed.

3.2.1 Multiple Catch Blocks

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following:

Syntax

```
try {
          body-code

        } catch (exception-classname1  variable-name1) {

          handler-code
        } catch (exception-classname2  variable-name2) {
```

```
  //Catch block
} catch(exception-classname3  variable-name3){

  //Catch block
}
```

Example Program No 3.2

```
class Multi_Catch
{
   public static void main (String args[ ])
 {
        int square_Array [ ] = new int [5];
        try        {
           for (int ctr = 1; ctr <=5; ctr++)
  {
               square_Array [ctr] = ctr * ctr;
           }
           for (int ctr = 0; ctr <5; ctr++)
   {
               square_Array [ctr] = ctr / ctr;
           }
        }  catch (ArrayIndexOutOfBoundsException e)        {
           System.out.println ("Assigning the array beyond the upper
bound");        }
        catch (ArithmeticException e)        {
           System.out.println ("Zero divide error");
}
}
}
```

Output

Assigning the array beyond the upper bound

Explanation

We are using two catch clauses catching the exception arrayIndexOutOfBoundsException and ArithmeticException.

3.3 Generic Exception

There is also a generic exception class called the Exception class. This class is capable of catching any type of exception. So while catching the exception, if you do not want to specify the exception type, you can put the generic exception and still handle any kind of exception the exception coming up. Take a look at example given below, though the exception generated is the ArithmeticException, the Exception class is able to handle it.

Example Program No 3.3

```
class GenericException
{
   public static void main (String args[])
   {
      try
      {
         int num = 34, den = 0;
            int quot = num / den;
      }
      catch (Exception e)
      {
         System.out.println ("Error in the code");
       }
   }
}
```

Output

Error in the code

Explanation

The generic Exception class can also be used with multiple catch blocks. While using the Exception class with multiple catch blocks one must exercise caution so as to not to keep this class in the top most catch clause. Since Exception class is the superclass of all the exceptions, it tends to catch any kind of exception. Consequently, all other catch blocks below it will be unreachable and your code will not be able to compile. So if you want to catch various exceptions particularly along with the generic exception, the generic exception must be put in the last catch.

Nested Try Statements

Try blocks may be written independently or we can nest the try blocks within each other, i.e., keep one try-catch block within another try-block.

Syntax

```
try {
   // statements
   try     {
      // statements
   } catch (exceptionclassname1  variablename1 )    {
      // statements
   }
   // statements
}  catch (exceptionclassname2 variablename2){
   // statements
}
```

Example for Nested Try blocks 3.4

```
class Nested_Try  {
  public static void main (String args [ ] )     {
       try         {

           int a = Integer.parseInt (args [0]);
           int b = Integer.parseInt (args [1]);
           int quot = 0;
           try   {
              quot = a / b;
              System.out.println(quot);   }
           catch (ArithmeticException e)   {
              System.out.println("divide by zero");
           }

       }catch (NumberFormatException e) {
           System.out.println ("Incorrect argument type");
       }
  }   }
```

Output

If the argument contains a string than the number:
The output is: Incorrect argument type
If the second argument is entered zero:
The output is: divide by zero

Explanation

Consider the above example in which you are accepting two numbers from the command line. After that, the command line arguments, which are in the string format, are converted to integers. If the numbers were not received properly in a number format, then during the conversion a NumberFormatException is raised otherwise the control goes to the next try block. Inside this second try-catch block the first number is divided by the second number, and during the calculation if there is any arithmetic error, it is caught by the inner catch block.

3.4 Throw, Throws and Finally Keywords

Throw

If you want to throw an exception explicitly then you need to use the throw clause. All the system-defined exceptions are thrown automatically, but the user-defined exceptions must be thrown explicitly using the throw clause. All methods use the throw statement to throw an exception. The throw statement requires a single argument is that a throwable object. Throwable objects are instances of any subclass of the Throwable class.

Syntax

throw new ThrowableInstance ( );

The flow of execution stops immediately after the throw statement, any subsequent statements are not executed. The nearest enclosing try block is inspected to see a catch that matches the exception.

Example Program No 3.5

```
class Throw_clause
{
   public static void main (String args [ ])
   {
try
{
```

```
    throw new UserDefinedException( );
}catch (UserDefinedException e)
{
  System.out.printIn ("User defined exception caught");
}
```

Output

User defined exception is caught.

Explanation

The UserDefinedException is a class made specifically to handle an exception and it is encapsulating some specific kind of user defined behavior. This exception is raised explicitly using the throw clause. At the same time the catch clause is there ready to catch the same exception object.

Throws

The throws keyword in java programming language is applicable to a method to indicate that the method raises particular type of exception while being processed. A throws clause lists the types of exceptions that a method might know.The throws keyword in java programming language takes arguments as a list of the objects of type java.lang.Throwables class.

Syntax

```
Type method_name(parameter_list)throws exception_list
{
//body of the method
}
```

Example Program No 3.6

```
class ThrowsClass {
static void throwMethod ( ) throws ClassNotFoundException {
System.out.println (" In throwMethod ");
throw new ClassNotFoundException ( );
 }
public static void main (String args [ ]) {
try {
throwMethod ( );
} catch ( ClassNotFoundException ObjA) {
```

```
System.out.println (" throwMethod has thrown an Exception :" +ObjA);
}
} }
```

Output

In throwMethod
throwMethod has thrown an Exception

Explanation

When the throw method is called, the statement says that there might be a chance of an exception and the exception is ClassNotFoundException. When the throw the exception the catch statement catches the ClassNotFoundException.

Difference between throw and throws keywords

Whenever we want to force an exception then we use **throw** keyword. The **throw** keyword (note the singular form) is used to force an exception. It can also pass a custom message to your exception handling module. Moreover **throw** keyword can also be used to pass a custom message to the exception handling module i.e. the message which we want to be printed. For instance in the above example we have used

throw new MyException ("can't be divided by zero");

Whereas when we know that a particular exception may be thrown or to pass a possible exception then we use **throws** keyword. Point to note here is that the Java compiler very well knows about the exceptions thrown by some methods so it insists us to handle them.

We can alsouse **throws clause** on the surrounding method instead of **try and catch exception handler.** For instance in the above given program we have used the following clause which will pass the error up to the next level

Static int divide(int first, int second) throws MyException{

Finally Keyword

If an exception is thrown, finally block will execute even if no catch statements matches the exception. Finally is guaranteed to execute, even if no exceptions

are thrown. Finally block is an ideal position for closing the resources such as file handle or a database connection etc.

Example Program No 3.7

```
public class FinallyDemo {
  static void price(){
    try{
      System.out.println("inside procA");
      throw new RuntimeException("demo");
     } finally{
      System.out.println("procA's finally");     }
Public static void main(String args[]) {
try {
procA();
} catch (Exception e) {
System.out.println("Exception caught");
}  }
```

Output

inside procA
Exception Caught
ProcA's finally

Explanation

When procA method is called, inside the try clause an exception is thrown which is caught. Then finally block will be executed, though an exception is caught in the main method.

3.5 Java's Built-in Exceptions

Java defines several exception classes inside the standard package.

java.lang

Since java.lang is implicitly imported into all Java programs, most exceptions derived from RuntimeException are automatically available.

Java defines several other types of exceptions that relate to its various class libraries.

Table 3.1 Java's Unchecked RuntimeException Subclasses

Sno	Exception	Description
1	ArithmeticException	Arithmetic error, such as divide-by-zero.
2	ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
3	ArrayStoreException	Assignment to an array element of an incompatible type.
4	ClassCastException	Invalid cast.
5	IllegalArgumentException	Illegal argument used to invoke a method.
6	IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
7	IllegalStateException	Environment or application is in incorrect state.
8	IllegalThreadStateException	Requested operation not compatible with current thread state.
9	IndexOutOfBoundsException	Some type of index is out-of-bounds.
10	NegativeArraySizeException	Array created with a negative size.
11	NullPointerException	Invalid use of a null reference.
12	NumberFormatException	Invalid conversion of a string to a numeric format.
13	SecurityException	Attempt to violate security.
14	StringIndexOutOfBounds	Attempt to index outside the bounds of a string.

Table 3.2: Java Checked Exceptions Defined in java.lang.

Sno	Exception	Description
1	ClassNotFoundException	Class not found.
2	CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
3	IllegalAccessException	Access to a class is denied.
4	InstantiationException	Attempt to create an object of an abstract class or interface.
5	InterruptedException	One thread has been interrupted by another thread.
6	NoSuchFieldException	A requested field does not exist.
7	NoSuchMethodException	A requested method does not exist.

Creating own Exception Subclasses.

You can create your own exceptions in Java.

Keep the following points in mind when writing your own exception classes:

1) All exceptions must be a child of Throwable.

2) If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.

3) If you want to write a runtime exception, you need to extend the RuntimeException class.

Example Program No 3.8

```
class NumberRangeException extends Exception   {
 String msg;
   NumberRangeException()     {
      msg = new String("Enter a number between 20 and 100");
   }
 }

public class My_Exception {
 public static void main (String args [ ])     {
      try        {             int x = 10;

         if (x < 20 || x >100) throw new NumberRangeException( );
      }       catch (NumberRangeException e)
      {
         System.out.println (e);
      }
```

Output

Number Range Exception

Explanation

The exception class here is the NumberRangeException class. You must have noticed that this class is derived from the exception class. A variable 'msg' of String type is defined and in the constructor it is initialized with appropriate message. You are also overriding the function toString with statement 'System.out.println (e);' so that the caught exception object can convey some message when used in a print statement.

Multithreading

3. 6 Concepts of Multithreading

Background: Multitasking allow executing more than one tasks at the same time, a task being a program. In multitasking only one CPU is involved but it can switches from one program to another program so quickly that's why it gives the appearance of executing all of the programs at the same time.Multithreading is the mechanism in which more than one thread run independent of each other within the process. Multitasking allow processes (i.e. programs) to run concurrently on the program.

Example Running the spreadsheet program and you are working with word processor also. Multitasking is running heavy weight processes by a single OS.

3.6.1 Multithreading concept

Multithreading is a technique that allows a program or a process to execute many tasks concurrently (at the same time and parallel). It allows a process to run its tasks in parallel mode on a single processor system .In the multithreading concept, several multiple lightweight processes are run in a single process/ task or program by a single processor.

Example When you use a word processor you perform a many different tasks such as printing, spell checking and so on. Multithreaded software treats each process as a separate program.

In Java, the Java Virtual Machine (JVM) allows an application to have multiple threads of execution running concurrently. When a program contains multiple threads then the CPU can switch between the two threads to execute them at the same time.

Example Below diagram, two threads are being executed having more than one task. The task of each thread is switched to the task another thread.

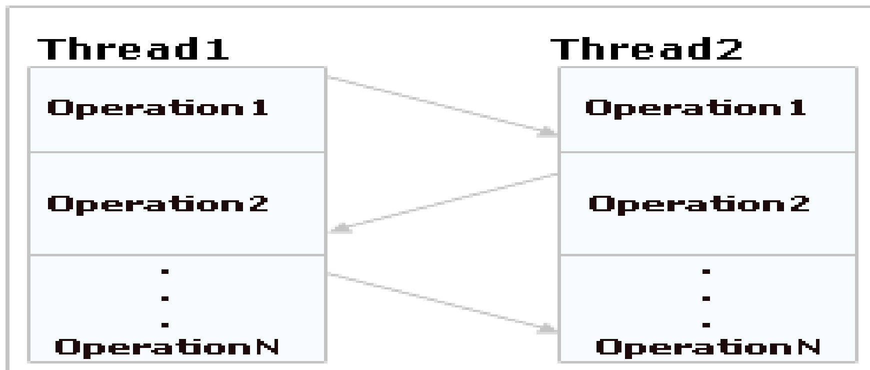


Fig 3.2: Switching of operation between two threads.

Table 3.3: Differences between process and thread

Sno	Process	Thread
1	Process is defined as the virtual address space and the control information necessary for the execution of a program	Threads are a way for a program to split itself into two or more simultaneously running tasks.
2	Processes do not share resources	A thread is contained inside a process and different threads in the same process share some resources.
3	Processes have their own address.	Threads share the address space of the process that created it
4	Processes have their own copy of the data segment of the parent process.	Threads have direct access to the data segment of its process
5	Processes must use inter-process communication to communicate with sibling processes.	Threads can directly communicate with other threads of its process
6	Processes have considerable overhead.	Threads have almost no overhead;
7	New processes require duplication of the parent process.	New threads are easily created

3.6.2 Thread Life Cycle

The following figure shows the states of a thread during thread life cycle and goes through 5 states: 1)when the thread create New State 2)after start() Runnable state
3) Running State 4) Wait/Block/Sleep State 5) Dead State.

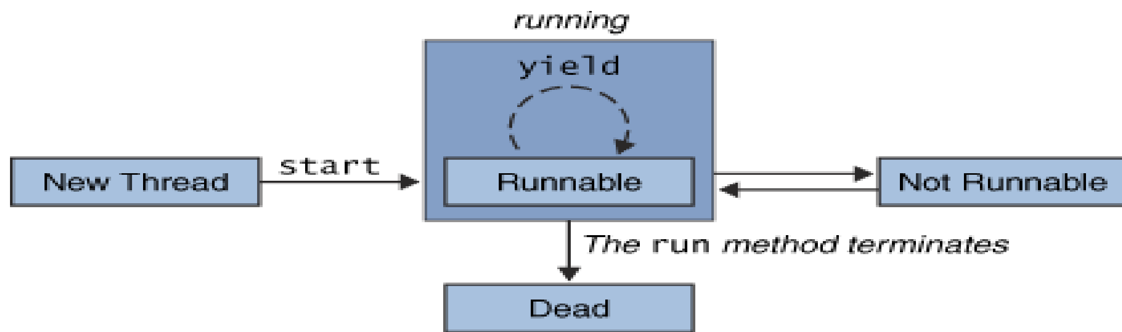


Fig 3.3: Life Cycle of aThread

3. 7 Creating Threads

Java's creators have graciously designed two ways of creating threads:

1. Extending a class.
2. Implementing an interface

3.7.1 Extending a class

The first method of creating a thread is to simply extend from the Thread class. Do this only if the class you need executed as a thread does not ever need to be extended from another class. The Thread class is defined in the package java.lang, which needs to be imported so that our classes are aware of its definition.

```
import java.lang.*;
public class Counter extends Thread
{
	public void run()
	{
	....
	}
}
```

3.7.2 Implementing an interface

The above example creates a new class Counter that extends the Thread class and overrides the Thread.run () method for its own implementation. The run() method is where all the work of the Counter class thread is done. The same class can be created by implementing Runnable:

```
import java.lang.*;
public class Counter implements Runnable
{
	Thread T;
	public void run()
	{
	....
	}
}
```

The only difference between the two methods is that by implementing Runnable, there is greater flexibility in the creation of the class Counter. In the above example, the opportunity still exists to extend the Counter class, if needed. The majority of classes created that need to be run as a thread will implement Runnable since they probably are extending some other functionality from another class.

3.8 Creating Multiple Threads using Thread class

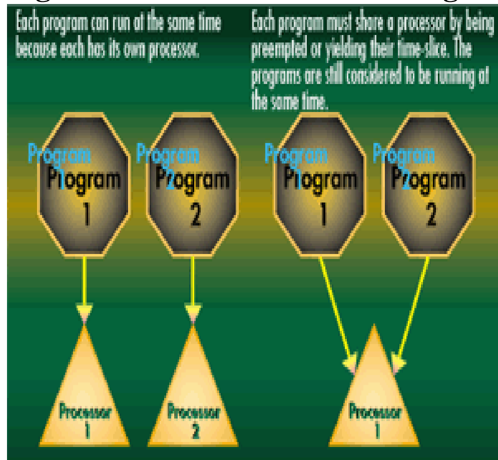
Background

The first method of creating a thread is to simply extend from the Thread class. Do this, only if the class you need executed as a thread does not ever need to be extended from another class. The Thread class is defined in the package java.lang, To create a new class that extends the Thread class and instantiate that class. The extending class must override the run() method and call start() method to begin execution of the thread.Inside run(), you will define the code that constitutes a new thread. It is important to understand that run() can call other methods, use other classes and declare variables just like the main thread. The only difference is the run() establishes the entry point for another, concurrent thread of execution within your program. This will end when run() returns.

Multi Threading Visual Look up

Fig-1) each application as running on its own processor.
Fig-2) each application share access to processors by preempting one application to let another one run.
Fig-3) Pieces of the same program can run concurrently

Fig-1 **Fig-2** **Fig-3**



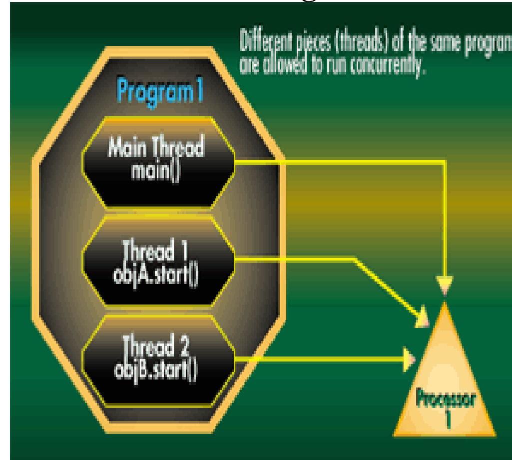


Fig: 3.4: Multithreading of The Same Program

Creating Multiple Threads using Thread class Example 3.9

```
public class MyThread extends Thread {
int n = 10, id;
public MyThread( int  ident) {
id = ident;
}
public void run() {
for(int i=0; i<n; ++i)
System.out.println("Hello from thread " + id);
} }
public class MyThreadApplic1 {
public static void main( String argv[] ) {
MyThread  myThread1, myThread2;
myThread1 = new MyThread( 1);
 myThread2 = new MyThread( 2);
myThread1.start();
myThread2.start();
System.out.println("Hello from main thread ");
}
}
```

Output

Hello from thread1
Hello from thread2
Hello from main thread

Creating Multiple Threads using Runnable Interface.

Example 3.10

```
public class RunnableClass implements Runnable {
int n = 10, id;
public RunnableClass( int  ident) {
id = ident;
}
public void run() {
for(int i=0; i<n; ++i)
System.out.println("Hello from thread " + id);
}}
public class MyThreadApplic2 {
public static void main( String argv[] ) {
RunnableClass ro1, ro2;
Thread MyThread1, MyThread2;
ro1 = new RunnableClass (1);
ro2 = new RunnableClass (2);
MyThread1 = new Thread (ro1);
MyThread2 = new Thread (ro2);
MyThread1.start ();
MyThread2.start ();
System.out.println ("Hello from main thread ");
} }
```

Output

Hello from thread 1
Hello from thread 2
Hello from main method

3.9 Methods Defined in Thread Class

Thread class is used to create a thread object; however, a class may implement the Runnable interface instead. Implementing the Runnable interface is useful when a class needs to be multi-threaded and also be derived from another class. This is an interface in java.lang. Package.

Table 3.4: Java.lang.Thread class methods

Sno	Method	Return Type	Description
1	currentThread( )	Thread	Returns an object reference to the thread in which it is invoked.
2	getName( )	String	Retrieve the name of the thread object or instance.
3	start( )	void	Start the thread by calling its run method.
4	run( )	void	This method is the entry point to execute thread, like the main method for applications.
5	sleep( )	void	Suspends a thread for a specified amount of time (in milliseconds).
6	join( )	void	This method and join(long millisec) Throws InterruptedException. These two methods are invoked on a thread. These are not returned until either the thread has completed or it is timed out respectively.
7	interrupt( )	void	The method interrupts the threads on which it is invoked.

3.10 Thread Priorities

In Java, thread scheduler can use the thread priorities in the form of integer value to each of its thread to determine the execution schedule of threads.Thread gets the ready-to-run state according to their priorities. The thread scheduler provides the CPU time to thread of highest priority during ready-to-run state. Priorities are integer values from 1 (lowest priority given by the constant Thread.MIN_PRIORITY) to 10 (highest priority given by the constant Thread.MAX_PRIORITY). The default priority is 5(Thread.NORM_PRIORITY).

Table 3.5: Thread priorities

Sno	Constant	Description
1	Thread.MIN_PRIORITY	The minimum priority of any thread (an int value of 1)
2	Thread.MAX_PRIORITY	The maximum priority of any thread (an int value of 10)
3	Thread.NORM_PRIORITY	The normal priority of any thread (an int value of 5)

The methods that are used to set the priority of thread shown below:

Table: 3.6: Methods Used In Threads

Sno	Method	Description
1	setPriority()	This is method is used to set the priority of thread.
2	getPriority()	This method is used to get the priority of thread.

Using Thread Priorities an example 3.11

```
class MyThread1 extends Thread
{
 MyThread1(String s){
 super(s);
 start();   }
 public void run(){
 for(int i=0;i<3;i++){
 Thread cur=Thread.currentThread();
 cur.setPriority(Thread.MIN_PRIORITY);
 int p=cur.getPriority();
 System.out.println("Thread Name  :"+Thread.currentThread().getName());
 System.out.println("Thread Priority  :"+cur);
 }
 }
 }

class MyThread2 extends Thread{
 MyThread2(String s){
 super(s);
 start();
}

public void run() {
for(int i=0;i<3;i++){
Thread cur=Thread.currentThread();
cur.setPriority(Thread.MAX_PRIORITY);
System.out.println("Thread Name  :"+Thread.currentThread().getName());
int p=cur.getPriority();
System.out.println("Thread Priority  :"+cur);
 }
 }
 }
```

```
public class ThreadPriority{
 public static void main(String args[]){
MyThread1 m1=new MyThread1("My Thread 1");
MyThread2 m2=new MyThread2("My Thread 2");
  }
}
```

Output

```
Thread Name: MyThread1
Thread Priority:0
Thread Name: MyThread2
Thread Priority:5
```

3.11 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. Synchronization means several methods can be executed simultaneously using the same object. But if you want to execute one method at a time that method can be declared as synchronized.

Example In the stack either the push or pop operation can be executed at one time for that you declared both methods as synchronized.

Why need Synchronization?

Synchronization and concurrency challenges posed by multiple views of a given resource.

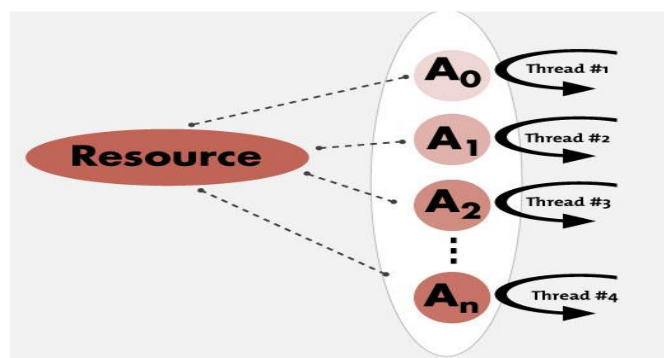


Fig 3.5: Synchronization between the Threads when they access the same resource.

Thread Synchronization on different processes

Mutex (Mutually Exclusive Lock)

Mutex is the thread synchronization object which allows accessing the resource only one thread at a time. Only when a process goes to the signaled state are the other resources allowed to access.

Semaphore (also called Monitor)

Semaphore is a thread synchronization object that allows zero to any number of threads access simultaneously. The mechanism that Java uses to support synchronization is the *monitor*. A monitor is like a building that contains one special room that can be occupied by only one thread at a time.

Synchronized Methods

The Java programming language provides two basic synchronization idioms: *synchronized methods* and *synchronized statements*.

Syntax of Synchronized method:

To make a method synchronized, simply add the synchronized keyword to its declaration:

```
void synchronized methodName() {       // body of method      }
```

Syntax of Synchronized statement:

```
synchronized(object) { // statements to be synchronized }
```

Implementing Synchronization

Locking

Locking follows a built-in acquire-release protocol controlled by the synchronized keyword. A lock is acquired on entry to a synchronized method or block and released on exit, even if the exit is the result of an exception. You cannot forget to release a lock. Locks operate on a per thread basis, not on a per-invocation basis. Java uses re-entrant locks means a thread cannot lock on itself.

Example for 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; }
```

Example for Synchronized Statements (block)

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

```
public void addName(String name) {

  synchronized(this)  {

  lastName = name;

  nameCount++;

 }

 nameList.add(name);

 }
```

3.12 Interthread Communication

Java provides a very efficient way through which multiple-threads can communicate with each-other. This way reduces the CPU's idle time i.e. a process where, a thread is paused running in its critical region and another thread is allowed to enter (or lock) in the same critical section to be executed. This technique is known as Interthreadcommunication.Methods for Interthread Communication:Java includes an elegant interprocess communication mechanism via the following methods:

1) wait ( ): This method tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ).

2) notify( ): This method wakes up the first thread that called wait( ) on the same object.

3)notify all( ): This method wakes up all the threads that called wait( ) on the same object. The highest priority thread will run first.These methods are implemented as final methods in Object, so all classes have them. All three methods can be called only from within a synchronized context.

These methods are declared within Object. Various forms of wait( ) exist that allow you to specify a period of time to wait.

Example Program for interthread communication: producer and consumer problem 1: 3.12

```
class Shared {
int num=0;
boolean value = false;
synchronized int get() {
 if (value==false)
try {
wait();
 }
catch (InterruptedException e) {
System.out.println("InterruptedException caught");
 }
System.out.println("consume: " + num);
value=false;
notify();
return num;
}
```

Example Program for interthread communication: producer and consumer problem 2 .3.13

```
synchronized void put(int num) {
  if (value==true)
  try {    wait();}
  catch (InterruptedException e) {
  System.out.println("InterruptedException caught");
    }
    this.num=num;
    System.out.println("Produce: " + num);
    value=false;
    notify();
    }  }
```

Example Program for interthread communication: producer and consumer problem 3.3.14

```
class Producer extends Thread {
  Shared s;

  Producer(Shared s) {
    this.s=s;
    this.start();
  }

  public void run() {
    int i=0;
    s.put(++i);
    }  }
```

Example Program for interthread communication: producer and consumer problem 4: 3.15

```
class Consumer extends Thread{
  Shared s;
  Consumer(Shared s) {
    this.s=s;
    this.start();
  }
  public void run() {
    s.get();
    }
}
```

Example Program for interthread communication :producer and consumer problem 5

```
public class InterThread{
  public static void main(String[] args)
  {
    Shared s=new Shared();
    new Producer(s);
    new Consumer(s);
  } }
```

3.13 Deadlocks

Deadlock describes a situation where two or more threads are blocked forever, waiting for each other.

If WorkA and WorkB if same instance of WorkA and WorkB are shared by both the Threads as shown below Thread Deadlock may happen, as depicted in diagram below:

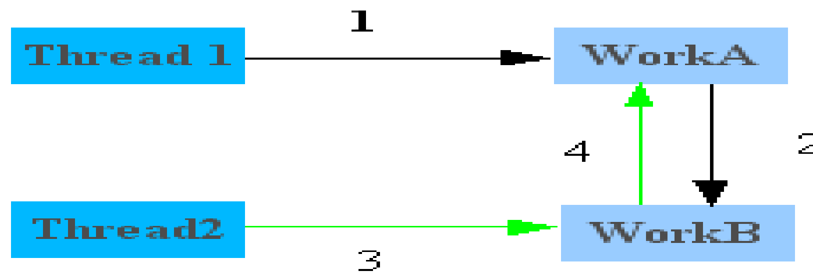


Fig: 3.6: Situation When The Deadlock Happens

Example for Deadlocks

Below diagram shows two threads having the Printing & I/O operations respectively at a time.But Thread1 need to printer that is hold up by the Thread2, likewise Thread2 need the keyboard that is hold up by the Thread1. In this situation the CPU becomes ideal and the deadlock condition occurs because no one thread is executed until the holdup resources are free.

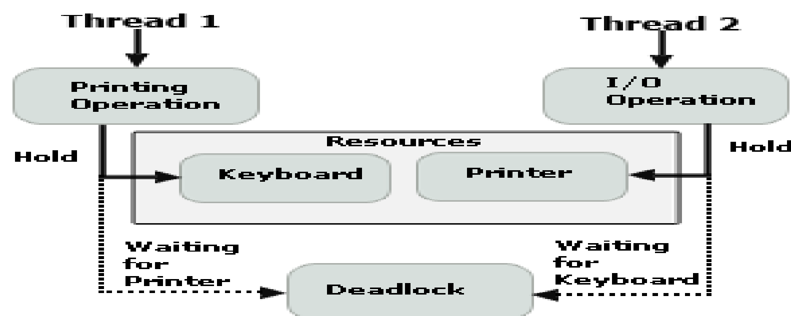


Fig: 3.7: Example Situation of Deadlock when they share The same resources

Example for Deadlock 3.16

```
public class DeadDemo{
  public static void main(String args[]){
     String s1="Dead";
  String s2="Lock";
  MyThread1 m=new MyThread1(s1,s2);
```

```
 MyThread2 m1=new MyThread2(s1,s2);
 }  }

class MyThread1 extends Thread{
 String s1;
 String s2;
 MyThread1(String s1, String s2){
 this.s1=s1;
 this.s2=s2;
 start();
 }
 public void run(){
   while(true){
   synchronized(s1){
    synchronized(s2){
   System.out.println(s1+s2);
     }
   }
  }
 }
 }
```

3.14 Thread groups

Every Java thread is a member of a *thread group*. Thread groups provide a mechanism for collecting multiple threads into a single object and manipulating those threads all at once, rather than individually. For example, you can start or suspend all the threads within a group with a single method call. Java thread groups are implemented by the ThreadGroup class in the java.lang package.

Threads are organized into thread groups. A thread group is simply a related collection of threads. For instance, for security reasons all the threads an applet spawns are members of a particular thread group. The applet is allowed to manipulate threads in its own group but not in others. Thus an applet can't turn off the system's garbage collection thread, for example. Thread groups are organized into a hierarchy of parents and children.

SUMMARY

1. Defintion of exception: An exception is a problem that arises during the execution of a program.
2. The Exception class has two main subclasses: IOException class and RuntimeException Class.
3. If you want to print your own message or the system message about the error then you write the part of the program which generate the error in the try{} block and catch the errors using catch() block.
4. You can write multiple catch blocks to one try block. And also multiple try blocks to the program.
5. Whenever we want to force an exception then we use **throw** keyword. Whereas when we know that a particular exception may be thrown or to pass a possible exception then we use **throws** keyword.
6. If an exception is thrown, finally block will execute even if no catch statements matches the exception. Finally is guaranteed to execute, even if no exceptions are thrown.
7. You can create your own exceptions in Java using the keyword new
8. A thread is an independent path of execution within a program. Many threads can run concurrently within a program
9. Multithreading is a technique that allows a program or a process to execute many tasks concurrently (at the same time and parallel).
10. There ar two ways of creating threads: 1. Extending a class. 2. Implementing an interface.
11. The only difference between the two methods is that by implementing Runnable, there is greater flexibility in the creation of the class
12. Thread priorities: In Java, thread scheduler can use the thread priorities in the form of integer value to each of its thread to determine the execution schedule of threads .
13. Synchronization means several methods can be executed simultaneously using the same object. But if you want to execute one method at a time that method can be declared as synchronized.
14. Interthreadcommunication: A process where, a thread is paused running in its critical region and another thread is allowed to enter (or lock) in the same critical section to be executed.

CHAPTER-4

APPLETS, EVENT HANDLING AND SWINGS

4.1 Applets

Definition of Applet

Applets are small applications that are accessed on an Internet Server, transported over the Internet, automatically installed, and run as part of a Web document.An **applet** is any small application that performs one specific task; sometimes running within the context of a larger program, perhaps as a plug-in. An applet is a special kind of Java program that a browser enabled with Java technology can download from the internet and run.

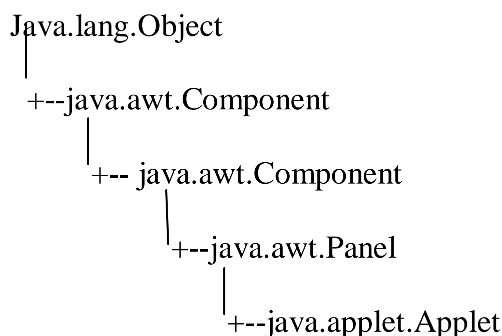
Other definition of Applet

Code written to execute under the control of a browser is called an applet. Applets can be used to provide dynamic user-interfaces, and a variety of graphical effects for web pages.Like applications, applets are created from classes. However, applets do not have a main method as an entry point, but instead, have several methods to control specific aspects of applet execution.

4.2 Applet Structure and Elements

The Java API Applet class provides what you need to design the appearance and manage the behavior of an applet. This class provides a graphical user interface (GUI) component called a Panel and a number of methods. To create an applet, you extend (or subclass) the Applet class and implement the appearance and behavior you want.

Applet Class



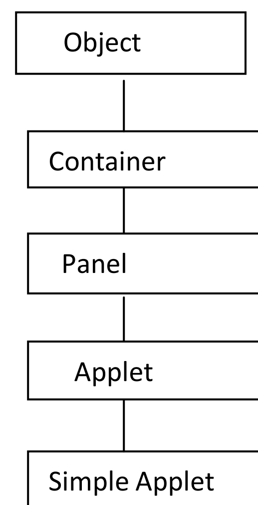


Fig: 4.1 Applet Structure

The SimpleApplet class extends Applet class, which extends the Panel class, which extends the Container class. The Container class extends Object, which is the parent of all Java API classes.

Difference between Application and Applet

The differences between an applet and an application are as follows:

1. Applets can be embedded in HTML pages and downloaded over the Internet whereas Applications have no special support in HTML for embedding or downloading.

2. Applets can only be executed inside a java compatible container, such as a browser or appletviewer whereas Applications are executed at command line by java.exe or jview.exe.

3. Applets execute under strict security limitations that disallow certain operations (sandbox model security) whereas Applications have no inherent security restrictions.

4. Applets don't have the main() method as in applications. Instead they operate on an entirely different mechanism where they are initialized by init(),started by start(),stopped by stop() or destroyed by destroy().

4.3 Applet Class

The Applet class provides the init, start, stop, destroy, and paint methods which you see in the next example applet. The SimpleApplet class overrides these methods to do what the SimpleApplet class needs them to do. The Applet class provides no functionality for these methods. An applet is a small program that is intended not to be run on its own, but rather to be embedded inside another application. The Applet class must be the superclass of any applet that is to be embedded in a Web page or viewed by the Java Applet Viewer. The Applet class provides a standard interface between applets and their environment

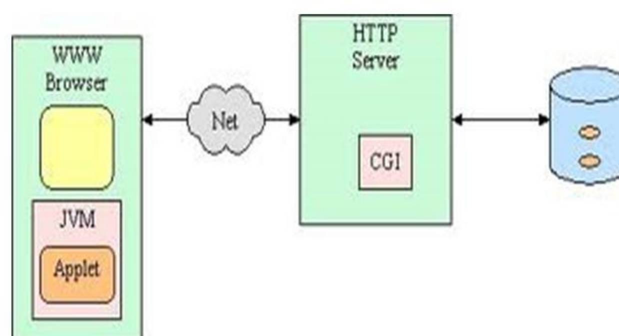
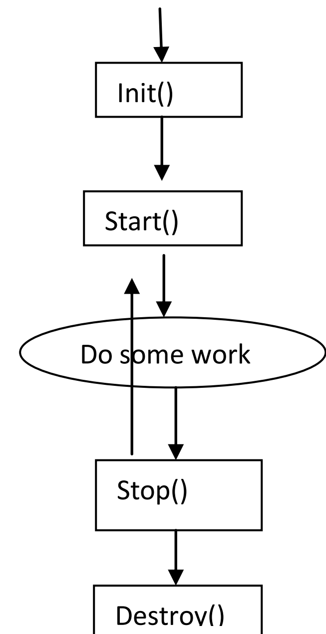


Fig 4.2 Use of Applet in the Real Time Applications

4.4 Methods of java applet

The Applet class provides a framework for applet execution, defining methods that the system calls when milestones occur. Milestones are major events in an applet's life cycle. Most applets override some or all of these methods to respond appropriately to milestones.



init(): This method is intended for whatever Initialization is needed for your applet. It is called after the *param attributes* of the applet tag.

start(): This method is automatically called after init method. It is also called whenever user returns to the page containing the applet after visiting other pages.

stop(): This method is automatically called whenever the user moves away from the page containing applets. You can use this method to stop an animation.

Destroy (): This method is only called when the browser shuts down normally. Thus, the applet can be initialized once and only once, started and stopped one or more times in its life, and destroyed once and only once.

Applet Display Methods

To output a string to an applet, use drawString(), a class of the Graphics is used.

Syntax

public void drawString(StringMsg, int x, int y)

The above method says: the string is to be output beginning at x, y. The method is typically called from within either update() or paint(). May Use setBackground() and setForeground() defined by Component class for the applet window. A good place to set foreground and background color is in the init(). The default foreground color is black, and default background color is light gray.

Example Program No 4.1

```
import  java.awt.*;
import java.applet.Applet;
public class HelloWorld extends Applet {
public void paint(Graphics g){
g.drawString(“Hello world!”, 30,30);
}}
```

Output

Explanation

This applet begins with two import statements. The first imports the Abstract window tool kit (AWT) classes. Applets interact with the user through the AWT, not through the console based I/O classes.The second import statement imports the applet package, which contains the class Applet. Every applet that you create must be a subclass of a Applet.

How to Run the Applet

To see the applet in action, you need an HTML file with the Applet tag as follows:

```
<HTML>
<BODY>
<APPLET CODE=SimpleApplet.class WIDTH=200 HEIGHT=100>
</APPLET>
</BODY>
</HTML>
```

The easiest way to run the applet is with appletviewer shown below where simpleApplet.html is a file that contains the above HTML code
appletviewer simpleApplet html

Example Program No 4.2

```
import java.applet.Applet;
import java.awt.Graphics; import java.awt.Color;
public class SimpleApplet extends Applet{
 String text = "I'm a simple applet";
 public void init() {
        text = "I'm a simple applet";
    }
  public void start() {
        System.out.println("starting...");
}
  public void stop() {
        System.out.println("stopping...");    }
  public void destroy() {
        System.out.println("preparing to unload...");    }
public void paint(Graphics g){
    System.out.println("Paint");
        g.setColor(Color.blue);
        g.drawRect(0, 0,
                   getSize().width -1,
                   getSize().height -1);
        g.setColor(Color.red);
        g.drawString(text, 15, 25);
  }
}
```

Important Note

1) To run an applet written with Java 2 APIs in a browser, the browser must be enabled for the Java 2 Platform.

2) If your browser is not enabled for the Java 2 Platform, you have to use appletviewer to run the applet or install Java Plug-in.

3) Java Plug-in lets you run applets on web pages under the 1.2 version of the Java VM instead of the web browser's default Java VM.

4.5 Parameter Passing to Applets

Java applet has the feature of retrieving the parameter values passed from the html page. So, you can pass the parameters from your html page to the applet embedded in your page. The param tag(<parma name="" value=""></param>)

is used to pass the parameters to an applet.Value of a parameter passed to an applet can be retrieved using *getParameter()* function.

Example

String strParameter = this.getParameter ("Message");
The function paint (Graphics g), prints the parameter value to test the value passed from html page.

Example Program No 4.3

```
import java.applet.*;
import java.awt.*;
public class appletParameter extends Applet {
private String strDefault = "Hello! Java Applet.";
public void paint(Graphics g) {
String strParameter = this.getParameter("Message");
if (strParameter == null)
strParameter = strDefault;
g.drawString(strParameter, 50, 25);
  }  }
```

HTML Code

```
<HTML>
<HEAD>
        <TITLE>Passing Parameter in Java Applet</TITLE>
         </HEAD>
              <BODY> This is the applet:<P>
                   <APPLET code="appletParameter.class" width="800"
height="100">
          <PARAM name="message" value="Welcome in Passing parameter in
java applet example.">
              </APPLET>
            </BODY>
</HTML>
```

Compile the program

***c:\>**javac appletParameter.java*

Output after running the program:

To run the program using appletviewer, go to command prompt and type *appletviewer appletParameter.html* Appletviewer will run the applet for you and and it should show output like **Welcome in Passing parameter in java applet example.** Alternatively you can also run this example from your favorite java enabled browser.

4.6 Event Handling

Event

An event is an object that describes a state change. Some of the activities that cause events to be generated are pressing a button, clicking the mouse, selecting an item in the list. An event may also be generated when a timer expires, software or hardware failure occurs, or an operation is completed.

Event source

A source is an object that generates an event this occurs when the internal state of that object changes. Sources may generate more than one type of event. A source must register listeners in order for the listeners to receive notification about the type of the event. Each type of the event has its own registration method.

Syntax Public void addTypeListener (TypeListener el)

Example public void addMouseListener (MouseListener me)

Event Listener

A listener is an object that is notify when an event occurs. It has two major requirements. First, it must have been registered with one or more sources. Second, it must implement methods.

Two Event Handling Mechanisms

There are two event handling mechanisms:

1) Java 1.0 (old) event handling is deprecated. (modified)

2) Java1.1 event handling is called Delegation Event Model. Presently implementing.

4.7 The Delegation Event Model

The modern approach to handling events is based on the delegation event model. A source generates an event and sends it to one or more listeners. The listener simply waits until it receives an event. Once received, the listener processes the event and then returns. Listeners must register with a source in order to receive an event notification.

Advantage The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events.

Event Delegation Model is based on three concepts:

1) The Event Classes: classes which broadcasts the events

2) The Event Listeners: classes which receive notifications of events

3) Event Objects: classes of objects which describes the event.

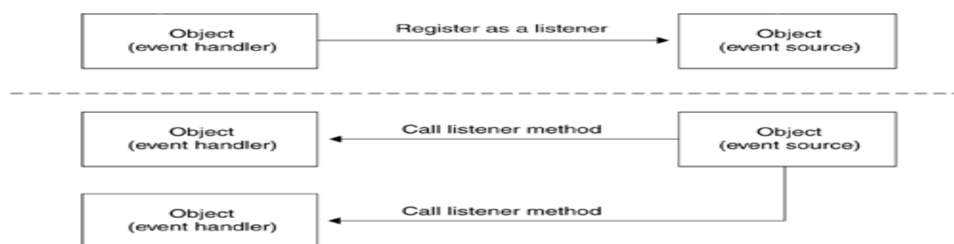


Fig 4.4: Delegation Event Model

4.8 Event Classes

Below shown event classes of package java.awt.event

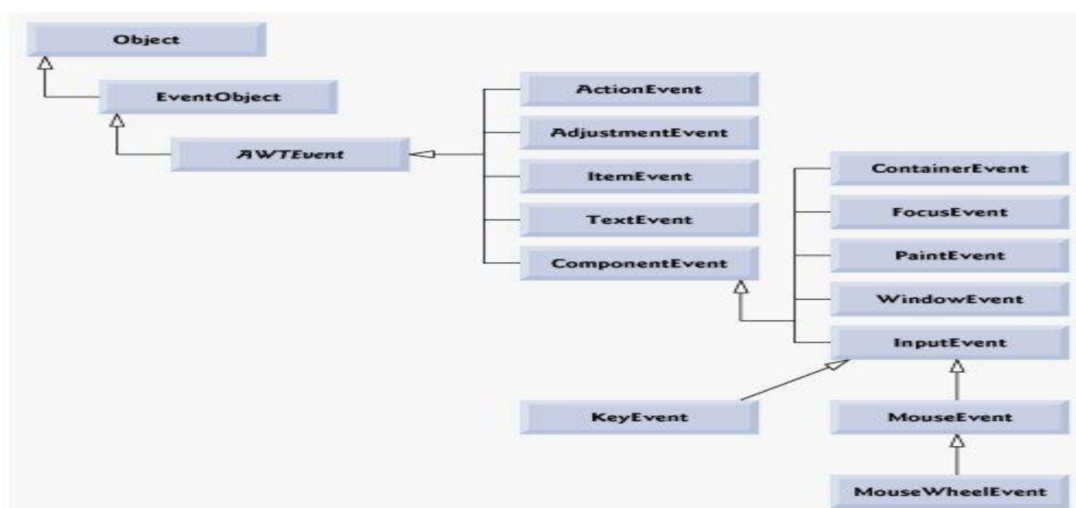


Fig 4.5: Event Classes Under Package java.awt.event

EventObject is the super class of all the events. AWT Event is the superclass of the AWTEvents that are handled by the delegation event model.

EventObject contains two methods: getSource( ) and toString( ).The getSource() returns the source of the event. toString() returns string equivalent of the event. The method of AWTEvent is int getID()

The description of the event classes

ActionEvent is generated whe following are clicked or selected.

Button : When the user clicks on a PushButton.

List : When an item in list box is double clicked.

MenuItem : When a MenuItem is selected.

TextField : When the user clicks the Enter key in a text box.

AdjustmentEvent is generated when the user adjusts the position of a scrollbar. Scrollbar is the only GUI control that receives the AdjustmentEvent.

ItemEvent is generated when an item is selected or deselected. Following components generate ItemEvents.

CheckBox : When the state of the CheckBox is changed.

CheckBoxMenuItem : When the state of a MenuItem is changed.

Choice : When the state of a ChoiceBox is changed.

List : When an item in list is selected or deselected.

TextEvent is generated when the contents of text component are changed. The components that generate TextEvent are TextArea and TextField.

FocusEvent This event is generated when a component gains or losses focus. Focus may be gained by bringing the mouse over a component (or by using the tab key). The component that has the focus receives all user keyboard events. Focus events are generated by objects of Component class and all its subclasses.

KeyEvent and MouseEvent are subclasses of abstract InputEvent class. Both these events are generated by objects of type Component class and its

subclasses. The KeyEvent is generated when the user presses or releases a key on the keyboard. The MouseEvent is generated when the user presses the mouse or moves the mouse.

WindowEvent are generated for the Window class and its subclasses. These events are generated if an operation is performed on a window. The operation could be closing of window, opening of window, activating a window etc.

PaintEvent is generated when a component needs to be repainted (for example when an application which is in front is closed and the component needs to be redrawn.) PaintEvents are internally handled by AWT, and cannot (and should not) be handled by you in the application.

ComponentEvent are also generated by objects of type Component class and its subclasses. This event is generated when a component is hidden, shown, moved or resized.

ContainerEvents are generated when a component is added or removed from a container. ComponentEvent and ContainerEvent are handled by AWT and are not normally handled by the user.

4.9 The Delegation Event Model working

The Communication between event source, event and listener interface implementation class(event handler).

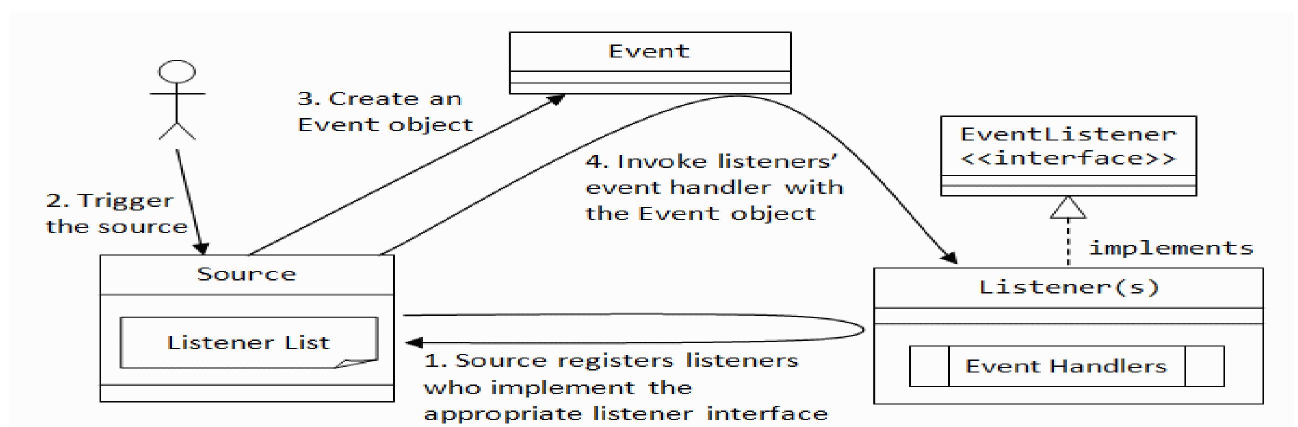


Fig 4.6: The Working of Delegation Event Model

1) The event source is the particular GUI component with which the user interacts.

2) The event object encapsulates information about the event that occurred, such as a reference to the event source and any event-specific information that may be required by the event listener for it to handle the event.

3)The event listener is an object that is notified by the event source when an event occurs; in effect, it "listens" for an event and one of its methods executes in response to the event.

4) A method of the event listener receives an event object when the event listener is notified of the event.

5)The event listener then uses the event object to respond to the event. The event-handling model described here is known as the **delegation event model,** it means an event's processing is delegated to a particular object (the event listener) in the application.Multiple listeners can register and to be notified of events of a particular type from a particular source. Also, the same listener can listen to notifications from different objects**.** In java below 1 and 2 points are most important.

1)Event Source must register listener, its format:
public void addTypeListener(TypeListener el)

2) TypeListener is a class Implements event listener interface methods.

Fig 4.7 Event Source Registering With More Than One Listeners

4.10 Event Sources

Event Source	Description
Button	Generates action events when the button is pressed
Checkbox	Generates item events when the check box selected or deselected
Choice	Generates item events when the choice is changed
List	Generates action events when item is double-clicked
MenuItem	Generates action events when menu item is selected.
Scrollbar	Generates adjustment events when the scroll bar is manipulated

Text components	Generates text events when the user enters a character
Window	Generates window events when a window is activated, closed or quit.

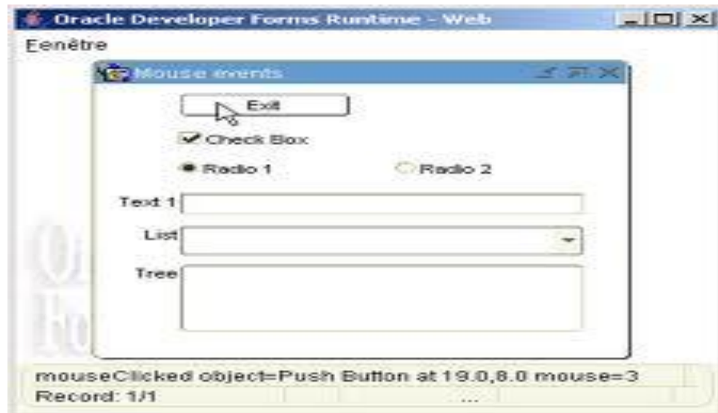


Fig: 4.8 The Applet Using All The Event Sources Like Button, Textbox

4.11 The Event Listeners Interfaces

Below Shown common event-listener interfaces of package java.awt.event
For example: when the user presses the Enter key in a TextField, the registered listener's actionPerformed method is called.

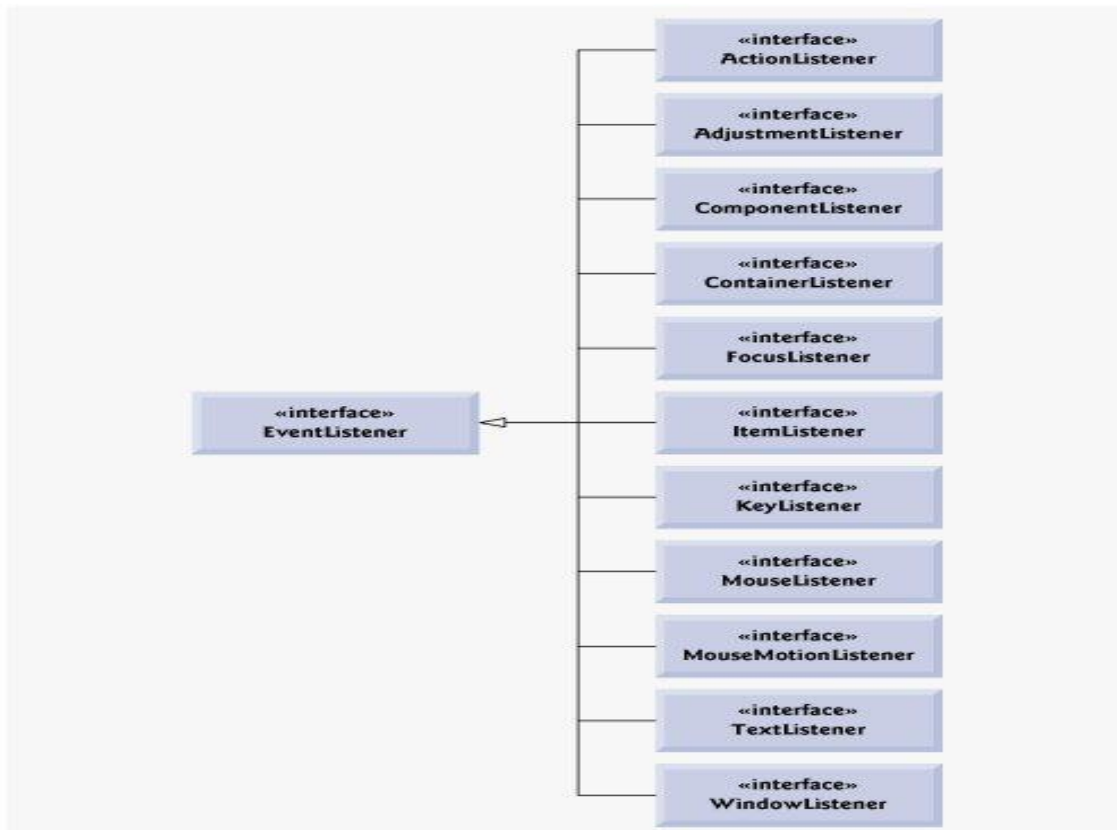


Fig 4.9: Event listener interface under java.awt.event

Interface	Inteface Methods	Addmethod
ActionListerner	actionPerformed(ActionEvent)	addActionListener()
AdjustementListener	adjustementValueChanged(AdjustementEvent)	addAdjustmentListener()
ComponentListner	componentHidden(ComponentEvent)	addComponentListener()
	componentMoved(ComponentEvent)	
	componentResized(ComponentEvent)	
	componentShown(ComponentEvent)	
ContainerListener	componentAdded(ComponentEvenet)	addContainerListener()
	componentRemoved(ComponentEvenet)	
FocusListener	focusGained(FocusEvent)	addFocusListener()
	FocusLost(FocusEvent)	
ItemListener	itemStateChanged(ItemEvent)	addItemListener
KeyListener	keyPressed(KeyEvent)	
	keyReleased(KeyEvent)	
	keytyped(KeyEvent)	
MouseListener	mouseClicked(MouseEvent)	addMouseListener()
	mouseEntered(MouseEvent)	
	mouseExited(MouseEvent)	
	mousePressed(MouseEvent)	
	mouseReleased(MouseEvent)	

MouseMotionListener	mouseDragged(MouseEvent)	addMouseMotionListener()
	mouseMoved(MouseEvent)	
TextListener	textValueChanged(TextEvent)	addTextListener()
WindowListener	windowActivated(WindowEvent)	addWindowListener()
	windowClosed(WindowEvent)	
	windowClosing(WindowEvent)	
	windowDeactivated(WindowEvent)	
	windowDeiconified(WindowEvent)	
	windowIconified (WindowEvent)	
	windowOpened(WindowEvent)	

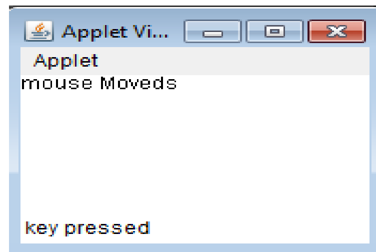
Table 4.11:GUI Component Interface and Methods

Example program to implement event listeners: 4.4

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="mousekeyEvents" width=500 height=500>
</applet>
*/
public class mousekeyEvents extends Applet implements MouseListener,
MouseMotionListener, KeyListener{
String msg="";
int mouseX=0, mouseY=0;
public void init(){
addMouseListener(this);
addMouseMotionListener(this);
addKeyListener(this);
requestFocus();
}
public void mouseClicked(MouseEvent me){
mouseX=0;
```

```
mouseY=10;
msg="mouse clicked";
repaint();
}
public void paint(Graphics g){
g.drawString(msg, mouseX, mouseY);
}
public void mouseExited(MouseEvent me){
msg="mouse Exited";
repaint();}
public void mouseDragged(MouseEvent me){
msg="mouse Dragged";
repaint();}
public void mousePressed(MouseEvent me){
msg="mouse Pressed";
repaint();}
public void mouseReleased(MouseEvent me){
msg="mouse Released";
repaint();}
public void mouseMoved(MouseEvent me){
msg="mouse Moved";
repaint();
showStatus(msg);}
public void mouseEntered(MouseEvent me){
msg="mouse Entered";
repaint();
}
public void keyPressed(KeyEvent ke){
showStatus("key pressed");
repaint();
}
public void keyReleased(KeyEvent ke){
showStatus("key pressed");
repaint();
}
public void keyTyped(KeyEvent ke){
msg+=ke.getKeyChar();
repaint();
}
}
```

Out put



Explanation

The mousekeyEvent class extends the Applet and implements MouseListener, MouseMotionListener and KeyListener interfaces. Inside init(), the applet registers itself as a listener for mouse and key events. This is done by using addMouseListener, addMouseMotionListener and KeyListener. The applet then implements all of the methods defined in the interfaces.

Requesting Repainting

Applet must quickly return control to the AWT run-time system.So to change particular information itself, we cannot make a loop in the paint method that repeatedly updates it.So, whenever your applet needs to update the information displayed in its window, it simply calls **repaint()**.

4.12 Adding and removing controls

To include a control in a window, you must add it to the window. To do this, you must create the instance of the control and add it to the window by calling add() method. To remove the button from the window use remove() method.

Handling Buttons:

Two constructors are:
Button()----------creates an empty button
Button(String str)-------- A button created with str string.
Methods:
void setLabel(String str)
String getLabel()

Example Program No 4.5

```
import java.awt.*;
import java.applet.*;
/*
```

```
<applet code=”ButtonDemo” width=300 height=300>
</applet>
*/
Public class ButtonDemo extends Applet implements ActionListener{
String msg=” ”;
Public void init() {
Button b1=new Button(“one”);
Button b2=new Button(“two”);
Button b3=new Button(“three”);
add(b1);
add(b2);
add(b3);
b1.addActionListener(this);
b2.addActionListener(this);
b3.addActionListener(this);
}
public void ActionPerformed(ActionEvent ae){
String s=ae.getActionCommand();
if(s.equals(“one”){
Msg=”you have pressed one”;
}
elseif(s.equals(“two”){
Msg=”you have pressed two”;
}
else(s.equals(“three”){
Msg=”you have pressed three”;
}
repaint();
}
public void paint(Graphics g){
g.drawString(msg,30,30);
}
}
```

Explanation

Each time a button is pressed, an action event is generated. This is sent to listeners that have previously registered. Each listener implements the ActionListener interface. That interface defines the actionPerformeed() method, which is called when an event occurs.

Handling CheckBoxes

CheckBox supports four constructors:
CheckBox()-------------------No label and unchecked
CheckBox(String str)------Label and unchecked
CheckBox(String str, boolean on)--------label and checked
CheckBox(String str, boolean on, CheckboxGroup cg)----label,checked and grouping the checkbox
CheckBox(String str, CheckboxGroup cg, boolean on)

Methods

Boolean getState()
Void setState(Boolean on)
String getLabel()
Void setLAbel(String str)

Example Program No 4.6

```
import java.awt.*;
import java.applet.*;
/*
<applet code="ButtonDemo" width=300 height=300>
</applet>
*/
Public class CheckboxDemo extends Applet implements ItemListener{
String msg=" ";
Public void init() {
Checbox b1=new Checkbox("one", true, null);
Checbox b2=new Checkbox("two", false);
Checbox b3=new Checkbox("three");
add(b1);
add(b2);
add(b3);
b1.addItemListener(this);
b2.addItemListener(this);
b3.addItemListener(this);
}
public void itemStateChanged(ItemEvent ie)
{
repaint();
}
public void paint(Graphics g)
```

```
{
msg=”state”
g.drawString(msg,30,30);
msg=”one”+b1.getState();
g.drawString(msg,30,40);
msg=”two”+b2.getState();
g.drawString(msg,30,50);
msg=”three”+b3.getState();
}
}
```

Explanation

Each time a checkbox is selected or deselected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item event notifications from that component. Each listener implements the ItemListener interface. That interface defines the itemStateChanged() method. An ItemEvent object is supplied as the argument to this method. It contains the information about the event. In the above example the first box will be checked and the second box is unchecked.In the checkbox only one item will be checked in the list. If you want multiple items to be checked, you can use checkbox group.

TextBox

Constructors

TextField()
TextField(int num chars)
TextField(string str)
TextField(string str,int numchars).

Methods

1.string getText()
2. void getText(string str)
3. string getselectedText()
4. void select(int startindex,int endindex)
5. void set editable(Boolean canedit)
6.boolean isEditable()
7. void set Echochar(char ch)
8. Boolean EchoChar Isset()
9. char getEchoChar()

Example Program No 4.7

```
import java.awt.*;
import java.applet.*;
/*
<applet code=”TextboxDemo” width=300 height=300>
</applet>
*/
public class CheckboxDemo extends Applet implements ActionListener{
String msg=” ”;
Public void init() {
TextField name=new TextField(10);
TextField pass=new TextField(6);
add(name);
add(pass);
add(b3);
name.addActionListener(this);
pass.addActionListener(this);
pass.setEchoChar(‘*’);
}
public void ActionPerformed(ActionEvent ae)
{
repaint();
}
public void paint(Graphics g)
{
msg=”name is”
msg+=name.getText();
g.drawString(msg,30,30);
msg=”pass is”
msg+=pass.getText();
g.drawString(msg,30,30);
}
}
```

Using Delegation Event Model Example

Here's an applet that puts up a single Button labeled "Beep." The Button is added to the applet. Then a BeepAction object is set to handle the Button's ActionEvents with the addActionListener () method.

Example Program No 4.8

```
import java.applet.*;

import java.awt.*;

import java.awt.event.*;

public class BeepApplet extends Applet {

public void init () {

// Construct the button

Button beep = new Button("Beep");

// add the button to the layout this.add(beep); // specify that action events sent
by this

// button should be handled by a new BeepAction object

beep.addActionListener(new BeepAction());

 }   }

import java.awt.*;

import java.awt.event.*;

 public class BeepAction implements ActionListener {

public void actionPerformed(ActionEvent e) {

 Toolkit.getDefaultToolkit().beep(); }}
```

4.13 AWT Graphics

AWT supports rich set of graphics methods

Drawing lines

To draw lines, the drawLine() method of the Graphics class is used.
void drawLine(int startx, int starty, int endx, int endy)
Example: g.drawLine(10, 10, 50, 50); }

Drawing rectangles

void drawRect(int top, int left, int width, int height)
This method takes four numeric attributes - the first two indicating the x/y starting point of the rectangle, the last two indicating the width and height of the rectangle.

Example:
//draw a rectangle starting at 100,100 width a width and height of 80
g.drawRect(100, 100, 80, 80); }

Filling a rectangle

By default a rectangle will have no color on the inside (it will just look like a box). You can use the fillRect() method to fill a rectangle. Set these values the same as you did for the drawRect() method to properly fill the rectangle.

Example

```
public void paint(Graphics g)
{
//draw a rectangle starting at 100,100 width a width and height of 80
g.drawRect(100, 100, 80, 80); g.fillRect(100, 100, 80, 80);
 }
```

Setting color

The rectangle is filled, but we didn't set a color for it! To do this, we will use the setColor() method.
g.setColor(Color.orange);

Drawing ovals

The drawOval() method takes four numeric attributes - the first two indicating the x/y starting point of the oval, the last two indicating the width and height of the oval. Fill an oval with the fillOval() method which also takes four numeric attributes indicating the starting position to begin filling and the height and width. Set these values the same as you did for the drawOval() method to properly fill the oval.

Example

g.setColor(Color.gray); //draw an oval starting at 20,20 with a width and height of 100 and fill it g.drawOval(20,20, 100, 100); g.fillOval(20,20, 100, 100); }

Displaying images

To display images, the Image class is used together with the Toolkit class. Use these classes to get the image to display. Use the drawImage() method to display the image.

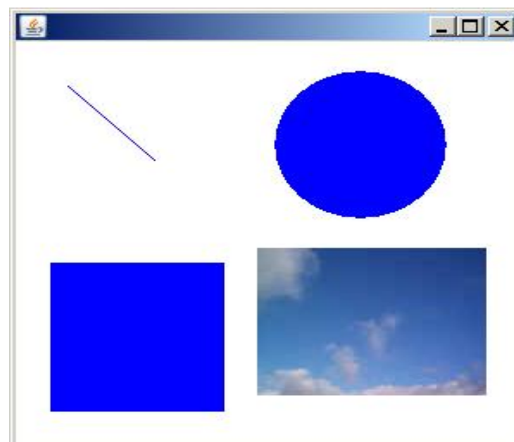
Example

Image img1 = Toolkit.getDefaultToolkit ().getImage ("sky.jpg"); //four attributes: the image, x/y position, an image observer g.drawImage (img1, 10, 10, this); }

Example Program No 4.9

```
import java.awt.*;
class GraphicsProgram extends Canvas{
public GraphicsProgram(){
setSize(200, 200);
setBackground(Color.white);
} public static void main(String[] argS){
GraphicsProgram GP = new GraphicsProgram();
aFrame = new Frame();
aFrame.setSize(300, 300); //add the canvas
aFrame.add(GP);
aFrame.setVisible(true); }
public void paint(Graphics g){
g.setColor(Color.blue);
g.drawLine(30, 30, 80, 80);
g.drawRect(20, 150, 100, 100);
g.fillRect(20, 150, 100, 100);
g.fillOval(150, 20, 100, 100);
Image img1 = Toolkit.getDefaultToolkit().getImage("sky.jpg");
g.drawImage(img1, 140, 140, this); } }
```

What it will look like:



AWT Classes

Abstract window toolkit is a package which includes all graphical class.

4.14 Swings –Introduction

Swing is the primary Java GUI widget toolkit. It is part of Sun Microsystems' Java Foundation Classes (JFC)- An API for providing a graphical user interface (GUI) for Java programs.

Swing was developed to provide a more sophisticated set of GUI components than the earlier Abstract Window Toolkit. Swing introduced a mechanism that allowed the look and feel of every component in an application to be altered without making substantial changes to the application code. The introduction of support for a pluggable look and feel allows Swing components to emulate the appearance of native components while still retaining the benefits of platform independence. AWT components are heavy-weight, whereas Swing components are lightweight.

Swing-Controls

Fig: 4.11JButton

Fig: 4.12JCheckBox

Fig: 4.13 JComboBox

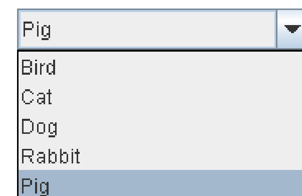


Fig: 4.14

JList Figs: 4.15 JMenu

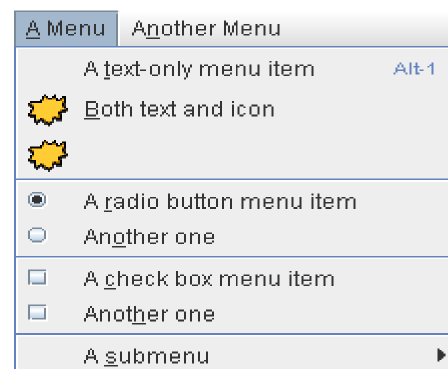


Fig: 4.16 JRadioButton

Fig 4.16 JSpinner

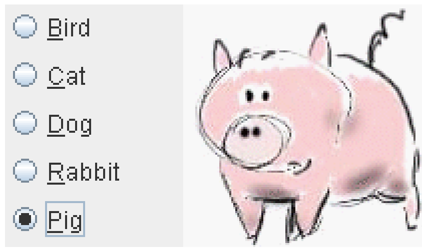
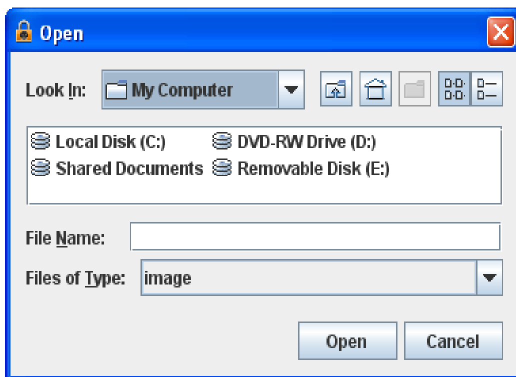


Fig: 4.17 JTextField

Figs: 4.18 JPasswordField

Fig: 4.19 JFileChooser

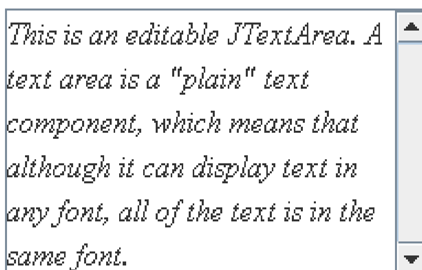
Fig 4.20 JTable

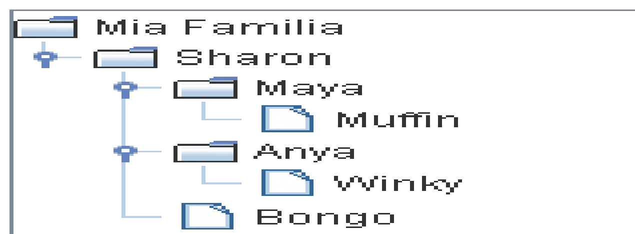


Host	User	Password	Last Modified
Biocca Games	Freddy	!#asf6Awwzb	Mar 16, 2006
zabble	ichabod	Tazb!34$fZ	Mar 6, 2006
Sun Developer	fraz@hotmail.co...	AasW541!fbZ	Feb 22, 2006
Heirloom Seeds	shams@gmail....	bkz[ADF78!	Jul 29, 2005
Pacific Zoo Shop	seal@hotmail.c...	vbAf124%z	Feb 22, 2006

Fig: 4.21 JTextArea

Fig: 4.22 JTree





The difference between the applet button and the swing button is that, swing button can have both an image and text. Also, the image can be changed as the state of the button changes.

Also the difference between the applet and JApplet is, when adding a component to an instance of JApplet, do not invoke the add() method of the

applet. Instead, add() for the content pane of the JApplet object. The content pane can be obtained via the method shown:

Container getContentPane ()

The add() method of container can be used to add a component to a content pane.

void add(comp)

Swing -Layout Managers :

All the components used so far are placed in a default browser. The layout manager automatically arranges the controls within the window using some specific algorithm. Several AWT and Swing classes provide layout managers for general use. Each Container object has a layout manager associated with it. A layout manager is an instance of any class that implements the LayoutManager interface. The layout manger is set by the setLayout(). In case no layout is specified, default layout manager is used.The default layout manger for applet is Flowlayout.

The syntax of setLayout method is :
SetLayout (LayoutManager layoutObj)

Fig: 4.23 BorderLayout

Fig: 4.24 BoxLayout

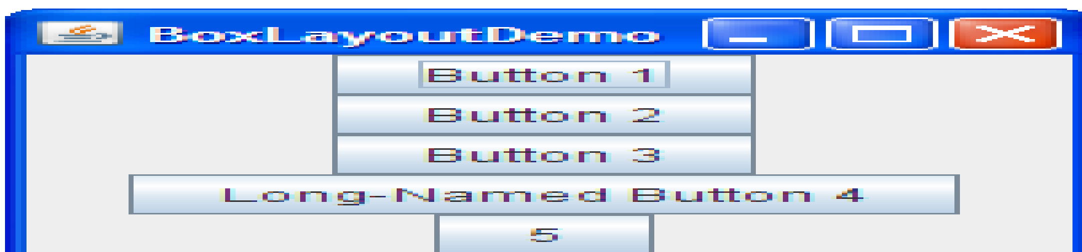


Fig: 4.25 CardLayout

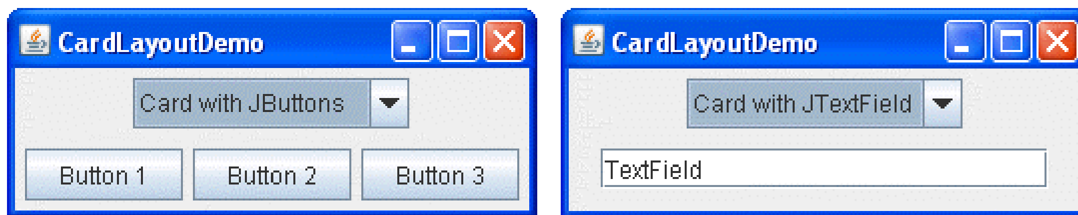


Fig: 4.26 FlowLayout

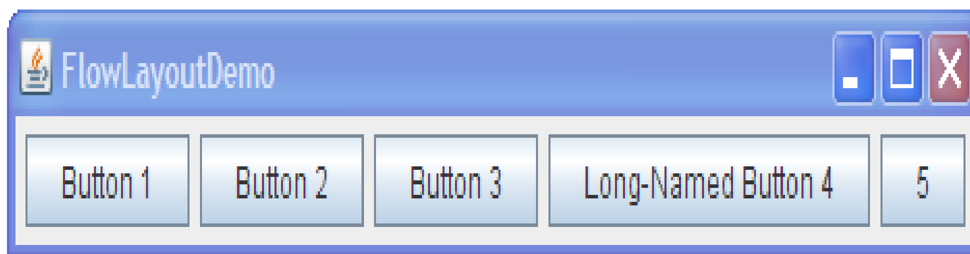


Fig: 4.27 GridBagLayout

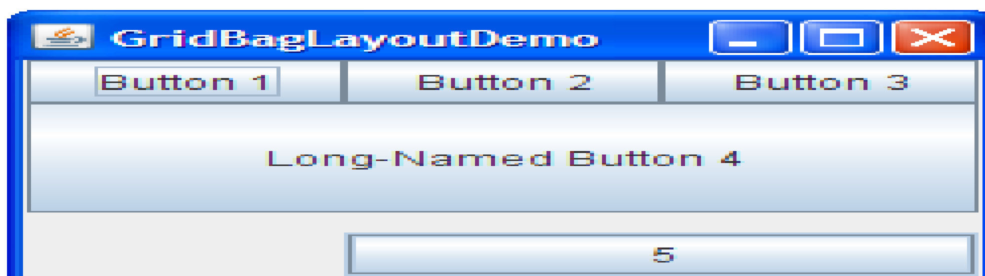


Fig: 4.28 GridLayout

Fig: 4.29 GroupLayout

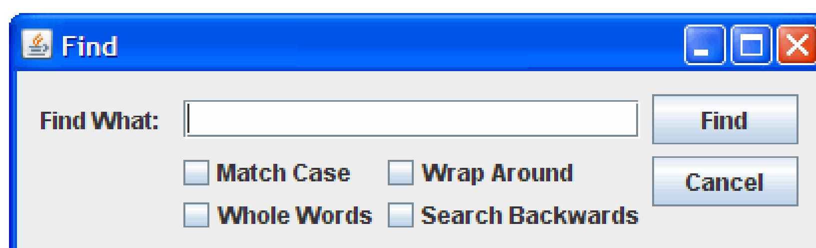


Fig: 4.30 SpringLayout

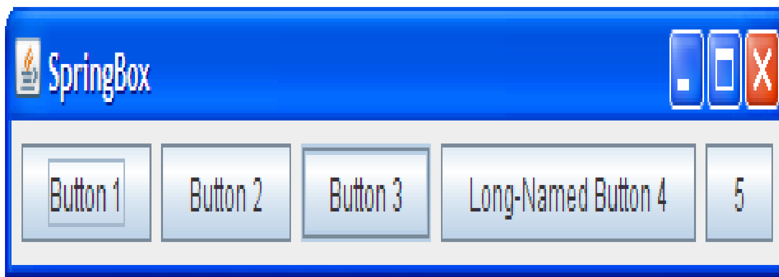
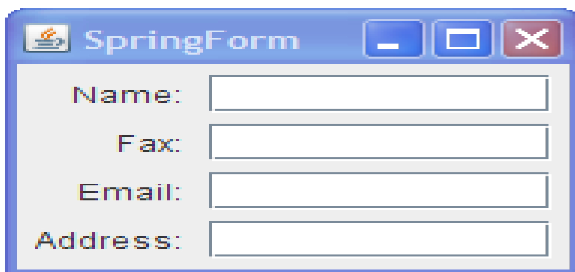


Fig: 4.31 SpringLayout



Swing-Layout Managers Usage

1. **Border Layout**: t o display a component in as much space as it can get.
2. **FlowLayout manager**: to display a few components in a compact row at their natural size. Consider using a JPanel to group the components
3. **GridLayout** : to display a few components of the same size in rows and columns.
4. **BoxLayout**: to display a few components in a row or column, possibly with varying amounts of space between them, custom alignment, or custom component sizes.
5. **SpringLayout**: to display aligned columns, as in a form-like interface where a column of labels is used to describe text fields in an adjacent column.
6. **GridBagLayout**: to have a complex layout with many components. Consider either using a very flexible layout manager

Flow Layout program example 4.10

```
import java.awt.*;
import java.applet.*;
/*
<applet code=”FlowLayoutDemo” width=300 height=300>
</applet>
*/
Public class FlowLayoutDemo extends Applet implements ItemListener{
String msg=” ”;
```

```
Public void init() {
Checbox b1=new Button("one", true, null);
Checbox b2=new Checkbox("two", false);
Checbox b3=new Checkbox("three");
FlowLayout fl=new FlowLayout(FlowLAyout.left,30,30)
setLayout(fl);
add(b1);
add(b2);
add(b3);
b1.addItemListener(this);
b2.addItemListener(this);
b3.addItemListener(this);
}
public void itemStateChanged(ItemEvent ie)
{
repaint();
}
public void paint(Graphics g)
{
msg="state"
g.drawString(msg,30,30);
msg="one"+b1.getState();
g.drawString(msg,30,40);
msg="two"+b2.getState();
g.drawString(msg,30,50);
msg="three"+b3.getState();
}}
```

Swing –Japplet

JApplet is the Swing equivalent of the AWT Applet class. The JApplet class is an extension of AWT Applet class. The interfaces implemented by the JApplet class are Accessible and RootPaneContainer. JApplet contains a JRootPane as its only child.The JApplet class can be used to create web page applets.

Package javax.swing

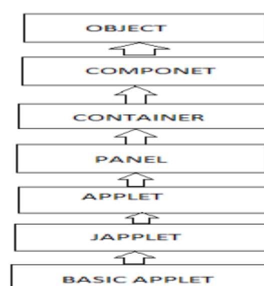


Fig 4.32 Swing Package

Example Program No 4.11

```
import java.awt.Graphics;
import javax.swing.JApplet;
public class HelloWorld extends JApplet {
public void paint( Graphics g ) {
super.paint( g );
g.drawString( "Hello World!", 25, 25 );
  }  }

<html>
<head>
   <title>Hello World</title>
</head>
<body>     <applet code="HelloWorld.class" height="300" width="300">
          Your browser is ignoring the applet tag.
   </applet> </body>
</html>
```

Swing-Icons

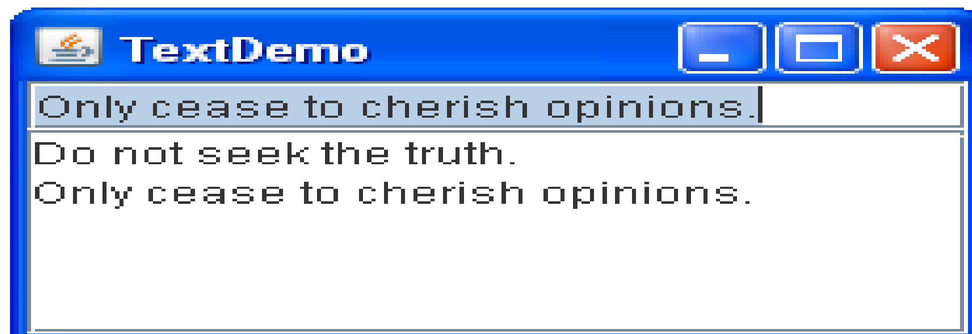
Many Swing components, such as labels, buttons, and tabbed panes, can be decorated with an *icon* -a fixed-sized picture. An icon is an object that adheres to the Icon interface. Swing provides a particularly useful implementation of the Icon. Interface: ImageIcon, which paints an icon from a GIF, JPEG, or PNG image. Here's a snapshot of an application with three labels, two decorated with an icon.

With the JLabel class, you can display unselectable text and images.
If you need to create a component that displays a string, an image, or both, you can do so by using or extending JLabel.The following picture introduces an application that displays three labels. The window is divided into three rows of equal height; the label in each row is as wide as possible

Swing-Text Fields

A text field is a basic text control that enables the user to type a small amount of text. When the user indicates that text entry is complete (usually by pressing Enter), the text field fires an action event. If you need to obtain more than one line of input from the user, use a text area.

The following example displays a basic text field and a text area. The text field is editable. The text area is not editable. When the user presses Enter in the text field, the program copies the text field's contents to the text area, and then selects all the text in the text field.



Example Program No 4.12

```
import javax.swing.JFrame;
public class Simple extends JFrame {
import javax.swing.*;
Public class SwingHello
{
public static void main(String[] a)
```

```
{
JFrame f = new JFrame("Hello Dear KLUians!");
f.setVisible(true); } }
```

We import the JFrame widget

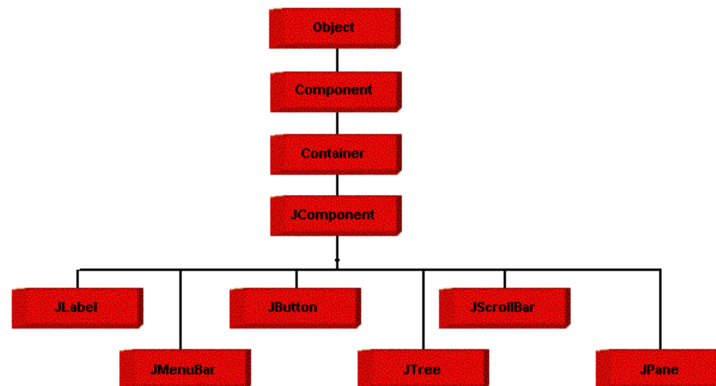


Fig: 4. Swing Class Hierarchy

Example Program No 4.13

```
import javax.swing.JFrame;
import javax.swing.JLabel;
//import statements
//Check if window closes automatically. Otherwise add //suitable code
public class HelloWorldFrame extends JFrame {
public static void main(String args[])
{
new HelloWorldFrame();
}
 HelloWorldFrame() {
JLabel jlbHelloWorld = new JLabel("Hello World"); add(jlbHelloWorld);
 this.setSize(100, 100); // pack(); setVisible(true); } }
```

SUMMARY

1. Definition of Applet: Applets are small applications that are accessed on an Internet Server, transported over the Internet, automatically installed, and run as part of a Web document.
2. The Applet class provides the init, start, stop, destroy, and paint methods.
3. We can pass the parameters from the HTML code to the applet, which will make some changes in the code and can be returned to the HTML.
4. Event: An event is an object that describes a state change. Examples are: pressing a button, clicking the mouse, selecting an item in the list.
5. Delegation event model. A source generates an event and sends it to one or more listeners. Listeners must register with a source in order to receive notification.
6. EventObject is the super class of all the events.
7. The different types of event classes present are ActionEvent, AdjustmentEvent, ItemEvent, TextEvent etc.
8. The various event sources are: button, checkbox, list, textbox etc.
9. The various interfaces present are: ActionInterface, Component Interface, Contianer Interface, FocusInterface etc.
10. We can add and remove the controls from the window using the method add(), remove().
11. Swing was developed to provide a more sophisticated set of GUI components than the earlier Abstract Window Toolkit.
12. The controls in the swing include the AWT controls such as button, label, textbox and have some controls Spinner and scroll panes.
13. The layout manager arranges the controls within the window using some specific algorithm.
14. The different types of layout managers are: border layout, flow layout, grid layout, box layout, spring layout.

CHAPTER-5

INTRODUCTON TO JAVA DATABASE CONNECTIVITY

5.1 What is JDBC?

Java Database Connectivity in short called as JDBC. It is a java API which enables the java programs to execute SQL statements.

5.2 Driver Types

Driver types are used to categorize the technology used to connect to the database. Some JDBC driver types are better suited for some applications than others.

JDBC drivers are divided into four types or levels. The different types of jdbc drivers are:

Type 1: JDBC-ODBC Bridge driver (Bridge)
Type 2: Native-API/partly Java driver (Native)
Type 3: All Java/Net-protocol driver (Middleware)
Type 4: All Java/Native-protocol driver (Pure)

Type 1 JDBC Driver

The Type 1 driver translates all JDBC calls into ODBC calls and sends them to the ODBC driver. ODBC is a generic API. The JDBC-ODBC Bridge driver is recommended only for experimental use or when no other alternative is available.

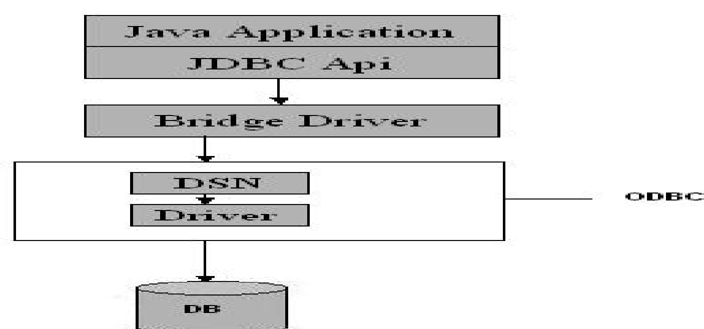


Fig.5.1 Type 1 Driver- the JDBC-ODBC Bridge

Type 2 JDBC Driver

Native-API/partly Java driver

The distinctive characteristic of type 2 jdbc drivers is that Type 2 drivers convert JDBC calls into database-specific calls i.e. this driver is specific to a particular database. Example: Oracle will have oracle native api.

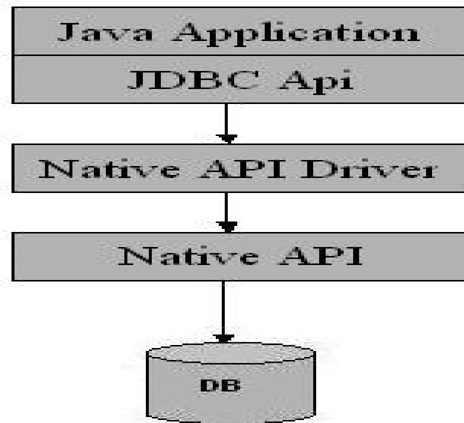


Fig.5.2 Type 2: Native api/ Partly Java Driver

Type 3 JDBC Driver

All Java/Net-protocol drivers

Type 3 driver database requests are passed through the network to the middle-tier server. The middle-tier then translates the request to the database. If the middle-tier server can in turn use Type1, Type 2 or Type 4 drivers.

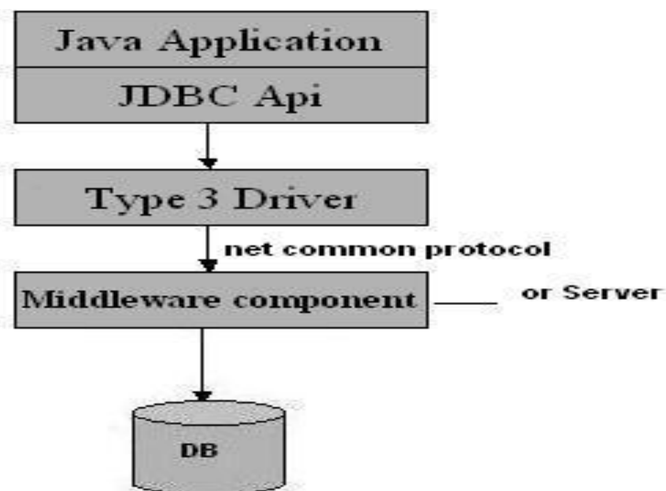


Fig. 5.3 Type 3: All Java/ Net-Protocol Driver

Type 4 JDBC Driver

Native-protocol/all-Java driver The Type 4 driver uses java networking libraries to communicate directly with the databasesever.

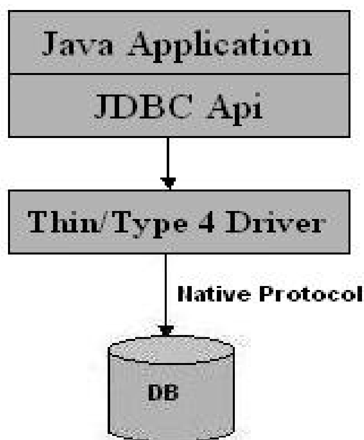


Fig. 5.4 Type 4: Native-protocol/all-Java driver

5.3 Establishing Connection to Database

The first thing to do is to install Java, JDBC and the DBMS on your working machines. If for example we want to interface with an Oracle database, we would need a driver for this specific database as well (type 2 driver).

A connection must be opened between our program (client) and the database (server) before a database can be accessed. This involves two steps:

- **Load the vendor specific driver**

  Since different DBMS's have different behavior, we need to tell the driver manager which DBMS we wish to use, so that it can invoke the correct driver.

  For example to connect to Oracle, the driver is loaded using the following line of code:

  ```
  Class.forName("oracle.jdbc.driver.OracleDriver)
  ```

- **Make the connection**

  Once the driver is loaded and ready for a connection to be made, you may create an instance of a Connection object using:

  ```
  Connection con = DriverManager.getConnection (
  ```

```
"jdbc:oracle:thin:@dbaprod1:1544:SHR1_PRD", username, passwd);
```

In above , The first string is the URL for the database including the protocol *(jdbc)*, the vendor *(oracle)*, the driver *(thin)*, the server *(dbaprod1)*, the port number *(1521)*, and a server instance *(SHR1_PRD)*. The username and passwd are *your* username and password, the same as you would enter into *SQLPLUS* to access your account.

5.4 Creating JDBC Statements.

A JDBC Statement object It is used to send your SQL statements to the DBMS. A JDBC Statement object is associated with an open connection, and not any single SQL Statement. An active connection is needed to create a Statement object. Using con object created above is used to create statement object in following line of code:

```
Statement stmt = con.createStatement() ;
```

JDBC PreparedStatement It is more convenient and efficient to use a PreparedStatement object for sending SQL statements to the DBMS.

PreparedStatements are also created with a Connection method. The following code shows how to create a parameterized SQL statement with three input parameters:

```
PreparedStatement prepareUpdatePrice = con.prepareStatement(
  "UPDATE Sells SET price = ? WHERE bar = ? AND beer = ?");
```

Before we can execute a PreparedStatement, we need to supply values for the parameters. This can be done by calling one of the setXXX methods defined in the class PreparedStatement. Most often used methods are setInt, setFloat, setDouble, setString etc. You can set these values before each execution of the prepared statement.

Continuing the above example, we would write:

```
prepareUpdatePrice.setInt(1, 3);
prepareUpdatePrice.setString(2, "Bar Of Foo");
```

5.5 JDBC API (Applications Programming Interface)

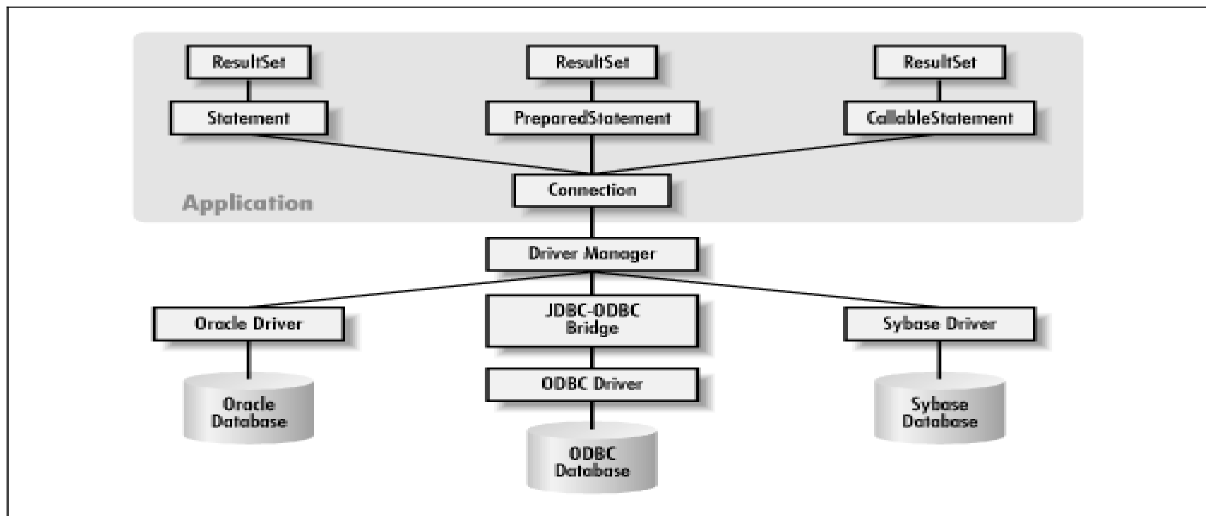
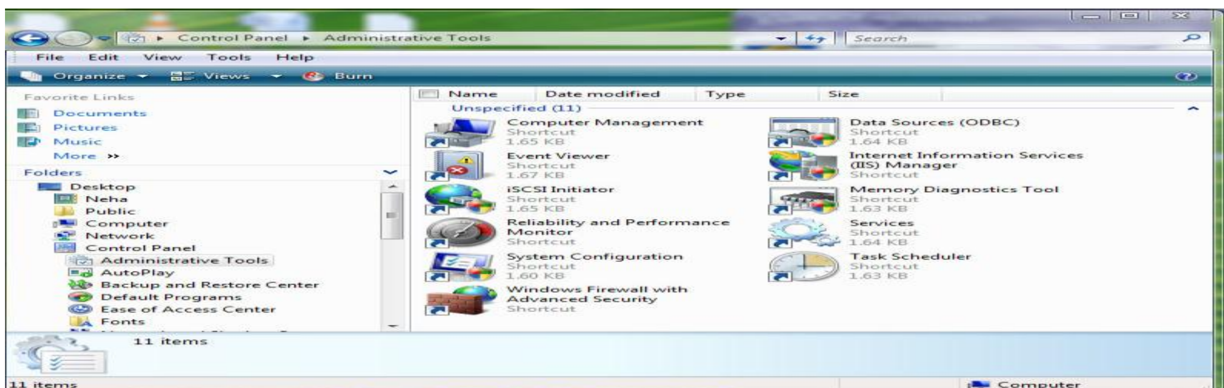


Fig: 5.5 JDBC API Classes used to connect , acesss and retrieve data from Database

5.6. Example Program using JDBC API.

OS: Windows Vista DBMS: MS Access 2007

1. Open control panel and go to Administrative tools.



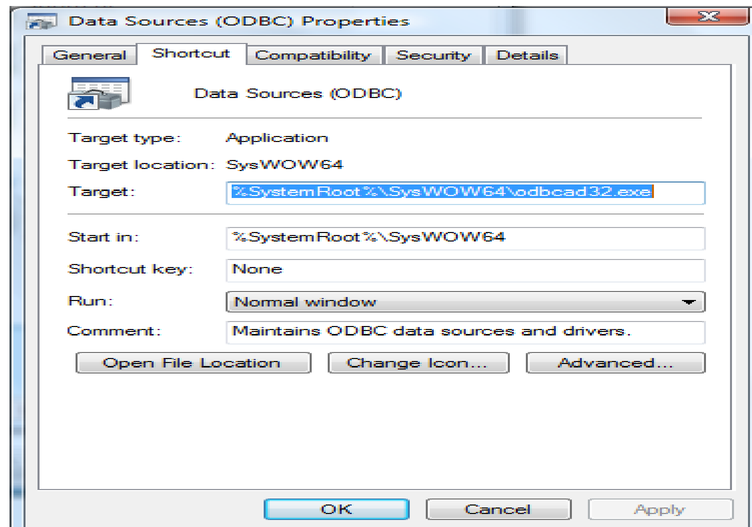
2. Right click on Data Sources (ODBC).

In properties, change the following and click 'OK'

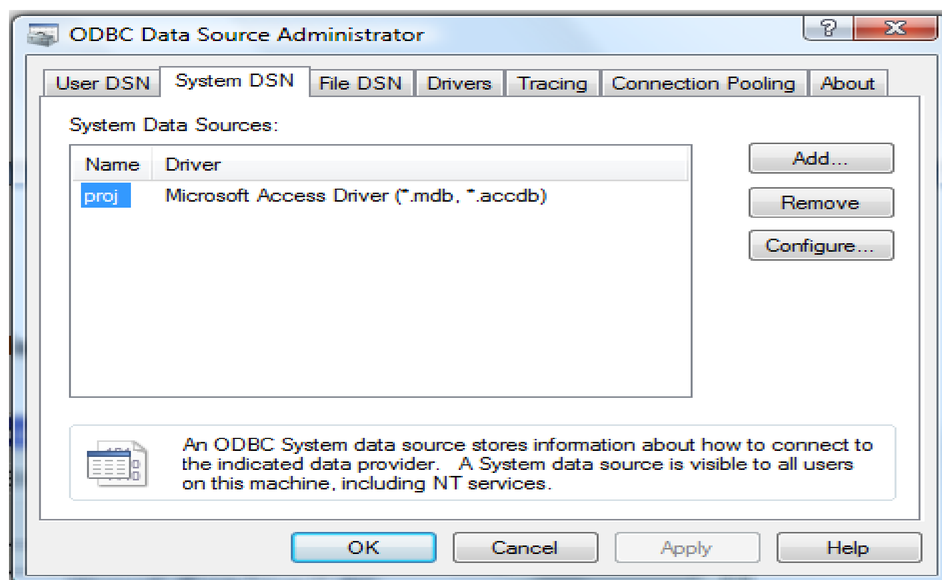
Target field to: %SystemRoot%\SysWOW64\odbcad32.exe

From:%SystemRoot%\System32\odbcad32.exe

Start In to: %SystemRoot%\SysWOW64

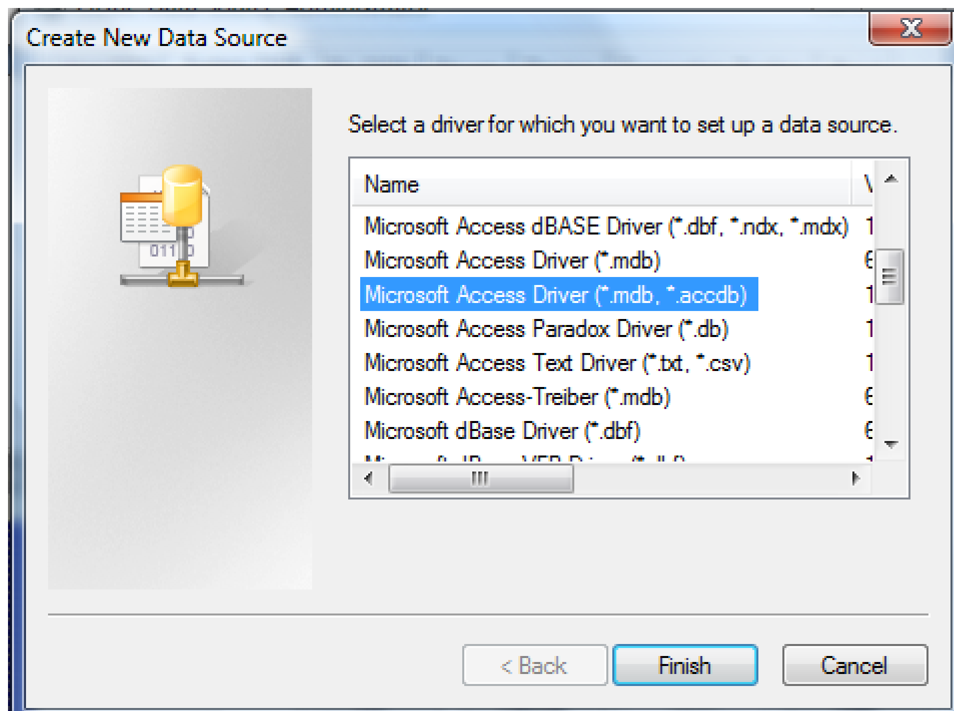


3. Double click on Data Sources (ODBC).Go to System DSN.



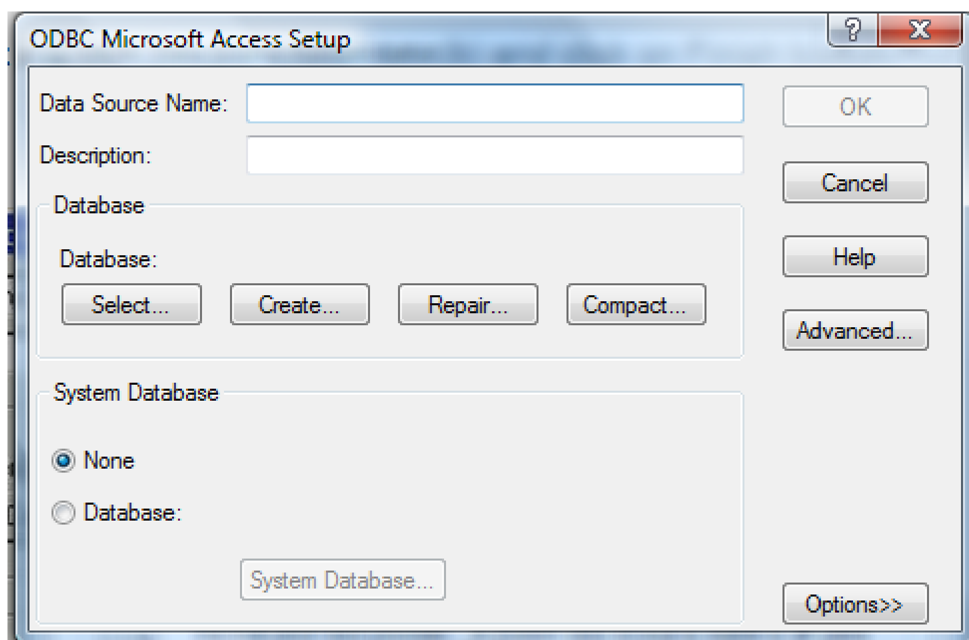
Click on Add button.

4. Then...



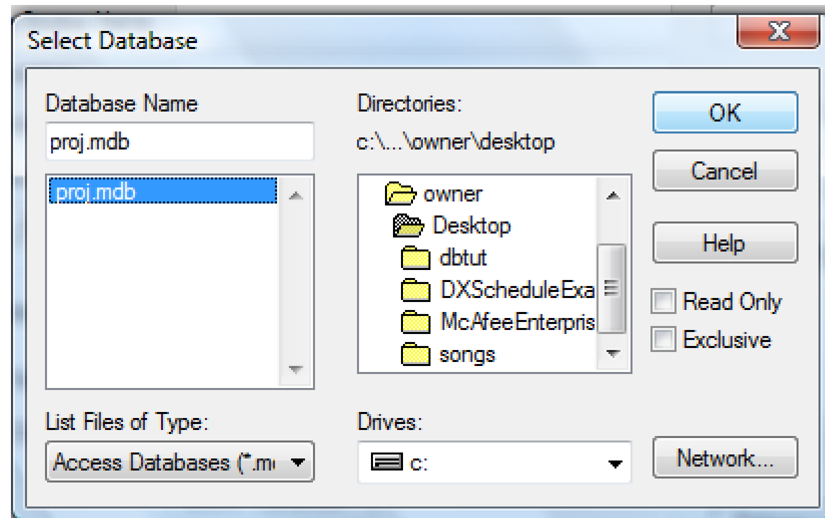
Click Microsoft Access Driver (*.mdb, *.accdb) and click on Finish button.

Then...



Insert your own Data Source Name (this is the name you will be using in the Java code to connect to the database, so ideally try to keep the database name and the DSN name to be the same) and click on Select button.

5. Then...



4.Choose your Database Access file like above and click OK button.

*** Note: before these procedures, we must have a Database Access file. We can make this file by using MS Access.

5. Now, we can test our JDBC program with MS Access.

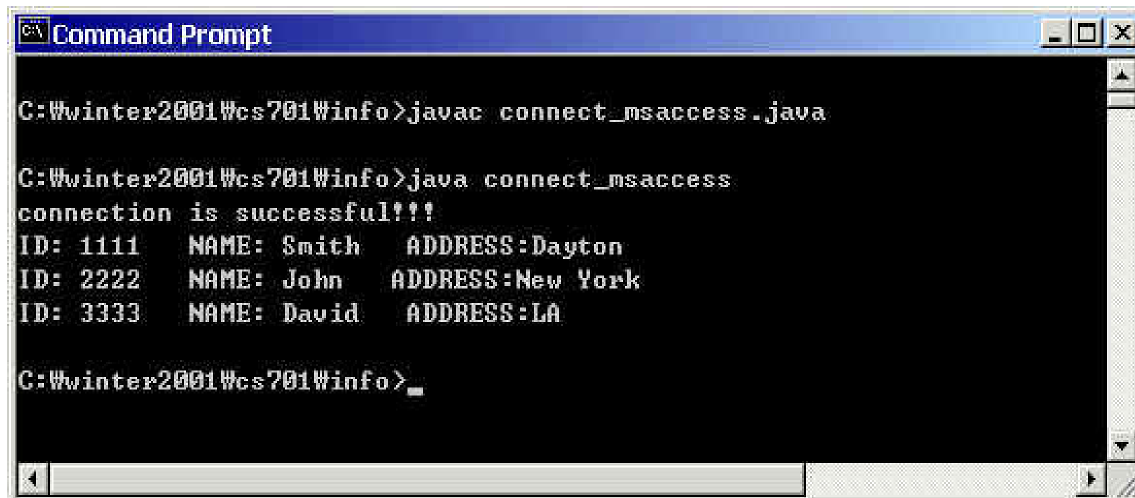
Here is a simple Java code that executes a SELECT statement.

Example Program No 5.1

```
import java.sql.*;
public class connect_msaccess
{
public static void main(String[] args)
{
int i;
Connection conn = null;
// register jdbc driver
try{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
} catch(ClassNotFoundException e) {
System.out.println(e);
}
// connect to DB
try{
conn = DriverManager.getConnection("jdbc:odbc:my_database");
```

```
} catch(SQLException se) {
System.out.println(se);
}
System.out.println("connection is successful!!!");
try{
String selectSQL = "select ID, NAME, ADDRESS from tb_address";
Statement stmt = conn.createStatement();
ResultSet rset = stmt.executeQuery(selectSQL);
while(rset.next()){
System.out.println("ID: " + rset.getString(1) + " NAME: " +
rset.getString(2) + " ADDRESS:" +
rset.getString(3));
}
stmt.close();
} catch(SQLException se) {
System.out.println(se);
} } }
```

8. Here is the result.



****

6. ADDITIONALS

6.1. Java interview Questions

1. What is an object?
2. What are the principles of object?
3. Explain the principles of oops
4. What is the difference between object oriented programming and object based programming?
5. What is static in java?
6. What if the main method is declared as private?
7. What if the static modifier is removed from the signature of the main method?
8. What if I write static public void instead of public static void?
9. Is JVM a compiler or an interpreter? – Interpreter. java interpreter convert the byte code(.class files) into sytem code(operating system understandale code) than it get execute.
10. What is UNICODE
11. What is the Vector class?
12. What is constructor and its importance in java?
13. What is the difference between a constructor and a method?
14. What is this keyword and super keyword in java?
15. What are different modifiers present in java? What is the default modifier?.
16. What are static methods?
17. What is the difference between final and finally keywords in java?
18. What is the difference between method overloading and overriding?
19. What is the difference between call by value and call by reference.?
20. What type of parameter passing does Java support?
21. What is the difference between static and non-static variables?
22. Can main method be declared final?
23. What will be the output of the following statement?
    System.out.println ("1" + 3);

24. What is the purpose of the System class?

25. Which class is extended by all other classes?

26. Can you call one constructor from another if a class has multiple constructors

27. For concatenation of strings, which method is good, StringBuffer or String ?
28. Explain the different forms of Polymorphism.
29. What is inheritance?
30. What are the types of inheritance?
31. What is the use of super keyword in inheritance?
32. What is abstract class?
33. What is interface and what is difference between abstract class and interface
34. What is dynamic method dispatch?
35. What is a package? What are different packages present in java?
36. Can we extend the interface?
37. What is a stream?
38. What does the I/O stream represent?
39. Can there be an abstract class with no abstract methods in it?
40. Can an Interface be final?
41. Can an Interface have an inner class?
42. Can I call a abstract method from a non abstract method ?
43. When can subclasses not access superclass members?
44. Can an inner class declared inside of a method access local variables of this method?
45. Can you call one constructor from another if a class has multiple constructors
46. What is Collection API?
47. What is an exception?
48. What is the difference between exception and error.
49. What are different types of exceptions

50. When a generic exception is present in the first catch block, can the specific exception like arithmetic exception in the second catch block is executed.
51. What is the difference between throw and throws.?
52. Can we throw our own exception in java?
53. What is thread. Difference between thread and process?
54. What are different states of a thread?
55. What are the ways to create the threads. What is best way to create it.?
56. What is default priority of a thread?
57. What is the mechanism defined by java for the Resources to be used by only one Thread at a time?
58. What are all the methods used for Inter Thread communication and what is the class in which these methods are defined?
59. what is a monitor object?
60. What state is a thread in when it is executing?
61. What is meant by deadlock. When does that situation arises?.
62. What is an applet?.
63. What is the difference between applet and application?
64. What are the methods of applet?
65. Which component subclass is used for drawing and painting?
66. What is the difference between the paint () and repaint () methods?
67. What is an event and when does it occur?
68. How can a GUI component handle its own events?
69. What is the difference between a window and a frame?
70. What are the different event sources?
71. What are the different event listener interfaces?
72. What is the layout manager?
73. What is the default layout manager?

****

6.2. Lab Manual covering 5 chapters with questions & exercises at the end of each program.

LAB PRACTICE MANUAL

LAB SESSION-I

Lab Session-1 covers the programs from Unit-1 (EXERCISE-1 TO 10) topics. PROGRAMS BASED ON LOGIC, TOPIC COVERAGE AND EXAMPLE APPLICATION.

EXERCISE-1

PASSING ARGUMENTS FROM COMMAND LINE TO JAVA MAIN METHOD

Aim: To know java main method to receive argument from command line

Learning Objectives:

- Use of Command Line Arguments.
- Use of Java main method.

Tool Required:

1.Netbeans java Editor.

```
public class commandLineArguments{
public static void main(String args[]) {
for(int i=0;i<args.length;i++){
System.out.println("commandline arugments are:"+args[i]);
}
}
```

Practice Exercise

1) Modify the above program which takes command line arguments as the following:

a)Java calculate Add 5 6
Output prints 11
b) java calculate sub 10 6
output prints 4

do it for add,sub,mul,div

Review Questions:

1)what is the use of main method ?
2)Why public used with main method?
3)what do you mean by commandline arguments?
4)what is data type of each commandline argument?
5)In System.out.println , System refers what?

EXERCISE-2

DISPLAY A GREET MESSAGE ACCORDING TO MARKS OBTAINED BY STUDENT, USING SUITABLE CONTROL STRUCTURE

Aim: To obtain marks of a student using control structures.

Learning Objectives:

- Use of control structures.

Tool Required:

1.Netbeans java Editor.

```
class StudentMarks{
public static void main(String args[]){
int marks = Integer.parseInt(args[0]);
switch(marks/10) {
case 10:
case 9:
case 8:
System.out.println("Excellent");
break;
case 7:
System.out.println("Very Good");
break;
case 6:
System.out.println("Good");
break;
case 5:
System.out.println("Work Hard");
break;
```

```
case 4:
System.out.println("Poor");
break;
case 3:
case 2:
case 1:
case 0:
System.out.println("Very Poor");
break;
default:
System.out.println("Invalid value Entered");
}  } }
```

Practice Exercise

1) Do the same program above using any other suitable control structure.

Review Questions

1) When default case executed?
2) What is the alternative control structure to switch statement?
3) Switch can take expression result value should be integer only if so, why, if not why?
4) Is case follows which data type value?

EXERCISE-3

MAGIC SQUARE TECHNIGUE USING ARRAYS AND USING CONTROL STRUCTURES

 Magic squares. An odd integer N from the command line and prints out an N-by-N magic square. The square contains each of the integers between 1 and N^2 exactly once, such that all row sums, column sums, and diagonal sums are equal.

```
4  9  2     11 18 25  2  9
3  5  7     10 12 19 21  3
8  1  6      4  6 13 20 22
            23  5  7 14 16
            17 24  1  8 15
```

One simple algorithm is to assign the integers 1 to N^2 in ascending order, starting at the bottom, middle cell. Repeatedly assign the next integer to the cell adjacent diagonally to the right and down. If this cell has already been assigned another integer, instead use the cell adjacently above. Use wrap-around to handle border cases.

Aim: To implement magic square technique using arrays and control structures.

Learning Objectives:

- Use of arrays.
- Use of control structures.

```
public class MagicSquare {
public static void main(String[] args) {
int N = Integer.parseInt(args[0]);
if (N % 2 == 0) System.out.println("N must be odd");
int[][] magic = new int[N][N];
int row = N-1;
int col = N/2;
magic[row][col] = 1;
for (int i = 2; i <= N*N; i++) {
if (magic[(row + 1) % N][(col + 1) % N] == 0) {
row = (row + 1) % N;
col = (col + 1) % N;
}
else {
row = (row - 1 + N) % N;
// don't change col
}
magic[row][col] = i;
}
// print results
for (int i = 0; i < N; i++) {
for (int j = 0; j < N; j++) {
if (magic[i][j] < 10) System.out.print(" ");
// for alignment
if (magic[i][j] < 100) System.out.print(" ");
// for alignment
System.out.print(magic[i][j] + " ");
}
System.out.println();
}
}
}
```

Practice exercise

1) Write a nested for loop that will print the following output:

```
                1
            1   2   1
        1   2   4   2   1
    1   2   4   8   4   2   1
1   2   4   8   16  8   4   2   1
        up to ..............128........................
```

Review Questions:

1) How java array different ordinary variable or c language array variable?
2) Array is java primitive data type true/false
3) Array can be used to store only primitive data type values if so, why ,if not why
4) Explain logic with an example
5) Why it is called magic square?

EXERCISE-4

2D ARRAY TO STORE WITH NAMES AND PRINT THEM EACH RANDOMLY

Aim: To use 2d array to store with names and print each name randomly.

Learning Objectives:

- Use of strings.

```
public class RandomStudent {
public static void main(String[] args) {
String[] names = { "Einstein", "Physics",
                   "Charles Babbage","Computers",
                   "Ramanujam”,"Mathematics",
                   "Amrthya Sen","Economics",
                   "C.V.Raman","Physics-Nobel Prize",
                   "F.Codd","DBMS"
                 };
int N = names.length;
int r = (int) (Math.random() * N);
System.out.println(names[r]);
}
}
```

Review Questions:
1)What are functions used with array in the above program, why it is used?
2) Why random function used?
3) which is the default package used by every java program?
4) In the above program names.length refers what?

EXERCISE-5

CREATE AN OBJECT AND CONTRUCTORS WITH SUITABLE EXAMPLE

Aim: To create an object and constructors using some examples.

Learning Objectives:

- Use of constructors.
- Use of Objects.

```
Class Cube {
int length;
int breadth;
int height;
public int getVolume() {
return (length * breadth * height);
}
Cube() {
this(10, 10);
System.out.println("Finished with Default Constructor of Cube");
}
Cube(int l, int b) {
this(l, b, 10);
System.out.println("Finished with Parameterized Constructor having 2 params
of Cube");
}
Cube(int l, int b, int h) {
length = l;
breadth = b;
height = h;
System.out.println("Finished with Parameterized Constructor having 3 params
of Cube");
```

```
}
public static void main(String[] args) {
Cube cubeObj1, cubeObj2; //cubeObj1,cubeObj2 are referencing object Cube
cubeObj1 = new Cube();
cubeObj2 = new Cube1(10, 20, 30);
System.out.println("Volume of Cube1 is : " + cubeObj1.getVolume());
System.out.println("Volume of Cube1 is : " + cubeObj2.getVolume());
}
}
}
```

Review Questions:

1)How many constructors method are there in above program?
2)How many object of cube created in total in the above program?
3) which is non-constructor method and why it is used?
4) what is the return values of cube?

EXERCISE-6

IMPLEMENT METHOD OVERLOADING WHERE ADD IS A METHOD CAN DO ADDING OF INTEGERS, FLOAT AND STRINGS

Aim: To implement method overloading where adding of integers, float and strings is done using add method.

Learning Objectives:

- Use of method overloading.
- Use of add method.

```
class MethodOverloading
 {
   void add(int a, int b) {            // 1 - A method with two parameters
     int sum = a + b;
      System.out.println(\"Sum of a+b is \"+sum);
    }
   void add(int a, int b, int c)       {
      int sum = a + b + c;
      System.out.println(\"Sum of a+b+c is \"+sum);
    }
   void add(double a, double b)            {
      double sum = a + b;
```

```
    System.out.println(\"Sum of a+b is \"+sum);
  }

  void add(String s1, String s2) {
    String s = s1+s2;
    System.out.println(s);
  }
 public static void main(String[] args)  {
 MethodOverloading obj = new MethodOverloading ();
 obj.add(1,2);
 obj.add(\"Life at \", \"?\");
 obj.add(11.5, 22.5);
 }
}
```

Practice exercise

1) write overload method to do the following:
   learn is function do following things:
   mathematics, computers, chemistry, physics

Review Questions:

1) what is method overloading?
2) Tell the method name overloaded , why overloaded in above program?
3) What us use of add method?
4) Advantages of method overloading ?

EXERCISE-7

DIFFERENT TYPES CASTING IN JAVA

Aim: To cast different types in java.

Learning objectives:

- Use of typecasting.

```
//Integer code1
i) public class TypeCasting1 {
public static void main(String arg[]) {
String s=”27”;
int i=Integer.parseInt(s);
System.out.println(i);
Float f=99.7f;
```

```
int i1=Integer.parseInt(f);
}
}
ii) //Integer code2
public class TypeCasting2 {
public static void main(String arg[]) {
String s=”27”;
int i=(int)s;
System.out.println(i);
int a=97;
String s=Integer.toString(a);
System.out.println(a);
}
```

Review Questions:

1) why do mean by casting?
2) What type of casting is performed in the above program?
3) Which method used to convert string to integer?
4) Why type conversion required?
5) What do mean of automatic type conversion?

EXERCISE-8

UTILIZE THE FEATURE OF INNER CLASSES IN JAVA

Aim: To utilize the feature of inner classes in java .

Learning Objectives:

- Use of inner classes.

```
Public class Class1 {
protected InnerClass1 ic;
public Class1() {
ic = new InnerClass1();
}
public void displayStrings() {
System.out.println(ic.getString() + ".");
System.out.println(ic.getAnotherString() + ".");
}
static public void main(String[] args) {
Class1 c1 = new Class1();
```

```
c1.displayStrings();
}
protected class InnerClass1 {
public String getString() {
return "InnerClass1: getString invoked";
}
public String getAnotherString() {
return "InnerClass1: getAnotherString invoked";
}
}
}
```

Review Questions:

1) what do you mean by innerclass?
2) In what condition do you use innerclass?
3) What is name of innerclass in the above program?
4) Difference of class and innerclass
5) In java give an example of inbuilt innerclass name?
6) Why Innerclass used and its purpose?

EXERCISE-9

DEMONSTRATE USING OF SOME STRING METHOD AVAILABLE IN JAVA

Aim: To know the demonstration of using of some string method of java.

Learning Objectives:

- Use of string method.

```
public class StringsDemo {
public static void main(String[] args) {
byte[] bytes = {2, 4, 6, 8};
char[] characters = {'a', 'b', 'C', 'D'};
StringBuffer strBuffer = new StringBuffer("abcde");
//Examples of Creation of Strings
String byteStr = new String(bytes);
String charStr = new String(characters);
String buffStr = new String(strBuffer);
System.out.println("byteStr : "+byteStr);
System.out.println("charStr : "+charStr);
System.out.println("buffStr : "+buffStr);
```

```
} }
```

Review Questions:

1) Is string what kind of datatype?
2)In How many do we can use string variable?
3)How do you create variable and reference variable?
4)List of some string methods availabele.
5)String belongs to which package.

EXERCISE-10

USING SOME STRING CLASS METHODS IN JAVA

Aim: To know the usage of some string class methods in java.

Learning Objectives:

- Use of string class methods.

```
import java.util.*;
import java.io.*;
import java.lang.String;
public class StringCount{
public static void main(String args[]){
String searchFor = "is";
String base = "This is the method";
int len = searchFor.length();
int result = 0;
if (len > 0) {
int start = base.indexOf(searchFor);
while (start != -1) {
result++;
start = base.indexOf(searchFor, start+len);
}  }System.out.println(result);
}}
```

Review Questions:

1)List out string method used in the above program
2)explain purpose of each string method used in above program

****

LAB SESSION-2

Lab Session-2 covers the programs from Unit-2 (exercise11to 21) topics. Programs based on the logic, topic coverage and example Application

EXERCISE-11

IMPLEMENT INHERITANCE CONCEPT TO VEHICLE

Aim: To implement inheritance concept to vehicle.

Learning Objectives:

- Use of inheritance.

```
class Vehicle {
	// Instance fields
	int noOfTyres; // no of tyres
	private boolean accessories; // check if accessories present or not
	protected String brand; // Brand of the car
	// Static fields
	private static int counter; // No of Vehicle objects created
	// Constructor
	Vehicle() {
		System.out.println("Constructor of the Super class called");
		noOfTyres = 5;
		accessories = true;
		brand = "X";
		counter++;
	}
	// Instance methods
	public void switchOn() {
		accessories = true;
	}
	public void switchOff() {
		accessories = false;
	}
	public boolean isPresent() {
		return accessories;
	}
	private void getBrand() {
```

```
                    System.out.println("Vehicle Brand: " + brand);
          }
          // Static methods
          public static void getNoOfVehicles() {
                    System.out.println("Number of Vehicles: " + counter);
          }
}
class Car extends Vehicle {
          private int carNo = 10;
public void  printCarInfo() {
 System.out.println("Car number: " + carNo);
 System.out.println("No of Tyres: " + noOfTyres); // Inherited.
 //  System.out.println("accessories: "    + accessories); // Not Inherited.
 System.out.println("accessories: " + isPresent()); // Inherited.
 //       System.out.println("Brand: "     + getBrand());  // Not Inherited.
 System.out.println("Brand: " + brand); // Inherited.
 //  System.out.println("Counter: "    + counter);     // Not Inherited.
                    getNoOfVehicles(); // Inherited.
          }
}
public class VehicleDetails { //
          public static void main(String[] args) {
                    new Car().printCarInfo();
          }
}
```

Review Questions:

1) what relation between or among objects makes objects are inherited?
2) By inheritance what do we achieve?
3) Why do we write overriding methods?
4) How do you create an object of parent class in subclass?
5) Can we call subclass method from superclass
6) Can we assign subclass object to superclass reference variable?

EXERCISE-12

DEMONSTRATING SUPER KEYWORD WITH INHERITANCE CONCEPT WITH AN EXAMPLE

Aim: To demonstrate the super keyword with inheritance concept using an example.

Learning Objectives:

- Use of inheritance.

```
class Box {
	double width;
	double height;
	double depth;
	Box() {
	}
	Box(double w, double h, double d) {
		width = w;
		height = h;
		depth = d;
	}
	void getVolume() {
		System.out.println("Volume is : " + width * height * depth);
	} }
public class Matchbox extends Box {
	double weight;
	MatchBox() {
	}
	MatchBox(double w, double h, double d, double m) {
		super(w, h, d);
		weight = m;
	}
	public static void main(String args[]) {
		MatchBox mb1 = new MatchBox(10, 10, 10, 10);
		mb1.getVolume();
		System.out.println("width of MatchBox 1 is " + mb1.width);
		System.out.println("height of MatchBox 1 is " + mb1.height);
		System.out.println("depth of MatchBox 1 is " + mb1.depth);
		System.out.println("weight of MatchBox 1 is " + mb1.weight);
	}
}
```

Review Questions:

1)can we assign superclass object to subclass reference variable?
2) what relation does exist between subclass and superclass in the above program?
3) what is the use of super method ? why it is used in above program
4) can subclass access superclass variable ? if yes how? Or no why?

5) can superclass access subclass variable ? if Yes how? Or no why?

EXERCISE-13

IMPLEMENT METHOD OVERRIDING WITH ANIMAL OBJECT AS AN EXAMPLE

Aim: To implement method overriding using animal object.

Learning Objectives:

- Use of method overriding

```
class Animal
 {
  int height=10;
  int weight=20;
 void talk()
  {    System.out.println(\"Animal talking\");
  }
  void food()
  {    System.out.println(\"Animal Eating\");
  }
 }
class Cat extends Animal
 { void talk()
   {
     System.out.println(\"meo... meo\");
   }
   void food()
   {     System.out.println(\"Drink MILK\");
   }
 }
class functionOverride
{
  public static void main(String[] args)
  {
   Animal a = new Animal();
   Cat c = new Cat();
   /* c.height=20 */;
   c.talk();
   c.food();
   System.out.println(\"Height \"+c.height);
```

```
  }
}
```

Review Questions:

1) what do you mean by method overriding?
2) Advantage of method overriding?
3) Why do we need to use method overriding?
4) Do we use overriding in inheritance relation exist among objects?
5) Which method overriding in the above program?

EXERCISE-14

ABSTRACT CLASS USING SHAPE OBJECT AS AN EXAMPLE

Aim: To abstract class using shape as an example.

Learning Objectives:

- To use abstract class.

```
abstract class Shape {
        public String color;
        public Shape() {
        }
        public void setColor(String c) {
                color = c;
        }
        public String getColor() {
                return color;
        }
        abstract public double area();
}
public class Point extends Shape {
        static int x, y;
        public Point() {
                x = 0;
                y = 0;
        }
        public double area() {
                return 0;
        }
        public double perimeter() {
```

```
            return 0;
    }
    public static void print() {
            System.out.println("point: " + x + "," + y);
    }
    public static void main(String args[]) {
            Point p = new Point();
            p.print();
    }
}
```

Review Questions:

1) what is abstract class?
2) Why do we use abstract class?
3) Difference of abstract class and class?
4) do we have abstract methods? If yes, how it can be declared?
5) do we create abstract class object ? if yes , how? Or no, why?
6) purpose of abstract class usage?

EXERCISE-15

IMPLEMENT CONCEPT OF POLYMOROHISM WITH MILK AS AN EXAMPLE

Aim: To implement concept of polymorphism using milk object as an example.

Learning Objectives:

- Use of polymorphism concept.

```
class Milk {
  void Prepare() {
    System.out.println("Boiling Milk to prepare next either Tea or Coffee");
  }
}

class Coffee extends Milk {
  void Prepare() {
    System.out.println("Prepare Coffee with Milk”);
  }
}

class Tea extends Milk {
```

```
  void Prepare() {
    System.out.println("Prepare Tea with Milk ");
  }
}

class DynamicDispatch {
  public static void main(String args[]) {
    DFMilkMilk M = new Milk(); // object of type Milk
Coffee C = new Coffee(); // object of type Coffee
    MilkTea0Tea T = new Tea(); // object of type Coffee
    LiqMilkMilk Mlk; // obtain a reference of type Milk
    LiMilkMilk = Mlk; // Mlk refers to an Milk object
    LMlkMlk.Prpeare(); // calls Milk's version of Prepare
    LiMilkMlk = C; // Mlk refers to a Coffee object
    LMlkMlk.Prepare(); // calls Coffee's version of Prepare
    LiqMMlk = T; // Liqd refers to a Tea object
    LiqkMlk.Prepare(); // calls Tea's version of Prepare
  }
}
```

Review Questions:

1)what do you mean by polymorphism?
2) How do you relate above program to polymorphism concept?
3)what is Dynamic method dispatch means?
4) Which method in above program dynamically dispatched?
5) Give a real life application of polymorphism.

EXERCISE-16

IMPLEMENT RUNTIME POLYMORPHISM WITH EMPLOYEE AS AN EXAMPLE

Aim: To implement runtime polymorphism using employee data.

Learning Objectives:

- To use runtime polymorphism.

```
class Employee {
  public void work() {
    System.out.println("I am an employee.");
```

```
  }
}
class Manager extends Employee {
  public void work() {
    System.out.println("I am a manager.");
  }
  public void manage() {
    System.out.println("Managing ...");
  }
}
public class Polymorphism {
  public static void main(String[] args) {
    Employee employee;
    employee = new Manager();
    System.out.println(employee.getClass().getName());
    employee.work();
    Manager manager = (Manager) employee;
    manager.manage();
  }
}
```

Review Questions:

1) what do you mean runtime polymorphism?

EXERCISE-17

INTERFACE TO OPERATING CAR OF DIFFERENT BRANDS

Aim: To implement an interface to operating car of different brands.

Learning Objectives:

- Use of interface.

```
public interface OperateCar {
   // constant declarations, if any
   // method signatures
   int turn(Direction direction,   // An enum with values RIGHT, LEFT
             double radius, double startSpeed, double endSpeed);
```

```
  int changeLanes(Direction direction, double startSpeed, double endSpeed);
  int signalTurn(Direction direction, boolean signalOn);
  int getRadarFront(double distanceToCar, double speedOfCar);
  int getRadarRear(double distanceToCar, double speedOfCar);
  // more method signatures
}
public class OperateBMW760i implements OperateCar {
  // the OperateCar method signatures, with implementation --
  // for example:
  int signalTurn(Direction direction, boolean signalOn) {
    //code to turn BMW's LEFT turn indicator lights on
    //code to turn BMW's LEFT turn indicator lights off
    //code to turn BMW's RIGHT turn indicator lights on
    //code to turn BMW's RIGHT turn indicator lights off
  }
  // other members, as needed -- for example, helper classes
  // not visible to clients of the interface
}
```

Review Questions:

1) what is an interface?
2) What is difference of class and interface?
3) Why interface does not contain code implementation?
4) When do you interface in applications?
5) How do you create objects for interface?
6) Where does the implementation of interface methods code exist?

EXERCISE-18

WRITE DATA TO A TEXTFILE

Aim: To implement writing of data to a text file.

Learning Objectives:

- Use of text file.

```
import java.io.*;
public class WriteFile{
   public static void main(String[] args) throws IOException{
     File f=new File("textfile1.txt");
     FileOutputStream fop=new FileOutputStream(f);
```

```
    if(f.exists()){
    String str="This data is written through the program";
        fop.write(str.getBytes());
        fop.flush();
        fop.close();
        System.out.println("The data has been written");
        }
        else
        System.out.println("This file is not exist");
  }
}
```

Review Questions:

1) what do mean by stream?
2) Which package need to import to work on files ?
3) What are classes of streams are used in above program?

EXERCISE-19

READ AN INPUT DATA AND CREATE A FILE

Aim: To implement reading data and create a file with read data.

Learning Objectives:

- To create a file using java stream classes.

```
import java.io.*;
public class FileWriter{
  public static void main(String[] args) throws IOException{
   BufferedReader in = new BufferedReader(new InputStreamReader(System.in
));
   System.out.print("Please enter the file name to create : ");
   String file_name = in.readLine();
   File file = new File(file_name);
   boolean exist = file.createNewFile();
   if (!exist)
   {
    System.out.println("File already exists.");
    System.exit(0);
   }
   else
   {
```

```
    FileWriter fstream = new FileWriter(file_name);
    BufferedWriter out = new BufferedWriter(fstream);
    out.write(in.readLine());
    out.close();
    System.out.println("File created successfully."); } }
}
```

EXERCISE-20

COPY OF FILE BY READING THE FILE CONTENT CHAR BY

Aim: To implement copying of file by reading the file.

Learning Objectives:

- Use of file.

```
import java.io.*;
public class rFile{
public static void main(String[] args throws IOException{
File inputFile=new File("/users/t/t.txt");
File outputFile=new File("/users/t/t2.txt");
FileReader in=new FileReader(inputFile);
FileWriter out=new FileWRiter(outputFile);
int c;
while((c=in.read())!=-1)
out.write(c);
in.close();
out.close();
}
}
```

EXERCISE-21

IMPLEMENT OBJECT SERIALIZATION

Aim: To implement object serialization using an example.

Learning Objectives:

- Use of object serialization.

```
import java.io.*;
 public class SerializingObject{
 public static void main(String[] args) throws IOException{
   BufferedReader in = new BufferedReader(new InputStreamReader(System.in
));
   System.out.print("Please enter File name : ");
   String file = in.readLine();
  System.out.print("Enter extension : ");
   String ext = in.readLine();
 String filename = file + "." + ext;
  File f = new File(filename);
 try{
  ObjectOutput ObjOut = new ObjectOutputStream(new FileOutputStream(f));
  ObjOut.writeObject(f);
  ObjOut.close();
  System.out.println("Serializing an Object Creation completed successfully.");
   }
catch(IOException e){
 System.out.println(e.getMessage());
 }
}
 }
```

Student Practice Exercises

1) Extend program A with the road bike and tandem bike.
2) Write a program to calculate area of different geometric shapes using polymorphic technique
3) Write a program with an interface single method returns result after calculation bank interest of saving account, deposit account, current account each have its own implementation logic.
4) Write a program to read date of students of five records contains Student name, student no, student total marks, student grade and calculate grade if total marks>500 is A grade else B grade .Write these all data to another file.

****

LAB SESSION-3

Lab session-3 covers the programs of Unit-3(Exercise-22to30) topics.
Programs based on Exceptions, Multi-threading concepts.

EXERCISE-22

EXCEPTION HANDLING

Aim: To use exception handling concepts of java.

Learning Objectives:

- Use of exception handling.

```
import java.io.*;
public class exceptionHandle{
  public static void main(String[] args) throws Exception{
    try{
      int a,b;
  BufferedReader in = new BufferedReader(new InputStreamReader(System.in)
);
      a = Integer.parseInt(in.readLine());
      b = Integer.parseInt(in.readLine());
    }
    catch(NumberFormatException ex){
      System.out.println(ex.getMessage() + " is not a numeric value.");
      System.exit(0);
    }
  }
   }
```

EXERCISE-23

EXCEPTIONS USING STRINGS

Aim: To use exceptions using strings.

Learning Objectives:

- Use of exceptions.

- Use of strings.

```
class MyException extends Exception {
  public MyException() {
  }
  public MyException(String msg) {
    super(msg);
  }
}

public class FullConstructors {
  public static void f() throws MyException {
    System.out.println("Throwing MyException from f()");
    throw new MyException();
  }
  public static void g() throws MyException {
    System.out.println("Throwing MyException from g()");
    throw new MyException("Originated in g()");
  }
  public static void main(String[] args) {
    try {
      f();
    } catch (MyException e) {
      e.printStackTrace();
    }
    try {
      g();
    } catch (MyException e) {
      e.printStackTrace();
    }}}
```

EXERCISE-24

PROGRAM THAT CATCHES THE EXCEPTION WHEN THE WORD CLIENT IS ENETRED INCORRECTLY

Aim: To use exception when the word client is entered incorrectly.

Learning Objectives:

- Use of exceptions.
- Use of strings.

```
import java.io.*;
```

```
// A Java application to demonstrate making your own Exception class
// This program catches the exception when the word "client" is
// entered incorrectly.

public class TestException {
  static String s = "";
    public static void main (String args[])
  {
    InputStreamReader is = new InputStreamReader(System.in);
    BufferedReader buf = new BufferedReader(is);
    System.out.println("Enter the word you cannot spell: ");
    try
    {
      s = buf.readLine();
    }
    catch (IOException e)
    {
      System.out.println("IOException was " + e.getMessage());
    }
        try
    {
      checkSpelling();   // this method throws SpellException
    }
    catch (SpellException se)   // but it is caught here
    {
      System.out.println("Spell exception was: " + se.getError());
    }
  } // end main
  // Check spelling of typed in word. Throw exception if wrong.
// Note how this method specifies that it throws such and such
// exception. Does not have to be caught here.
  private static void checkSpelling() throws SpellException
  {
    if (s.equalsIgnoreCase("client"))
      System.out.println("OK");
    else
      throw new SpellException("Cannot spell client");
   }
}  // end main class
// Custom exception class that descends from Java's Exception class.
class SpellException extends Exception
{
  String mistake;
// Default constructor - initializes instance variable to unknown
```

```
  public SpellException()
  {
    super();             // call superclass constructor
    mistake = "unknown";
  }
// Constructor receives some kind of message that is saved in an instance
variable.
  public SpellException(String err)
  {
    super(err);     // call super class constructor
    mistake = err;  // save message
  }
```

EXERCISE-25

SPECIAL BEHAVIOUR IN CATCH AND FINALLY BLOCKS

Aim: To use special behavior in catch and blocking finally.

Learning Objectives:

- Use of exceptions.

```
import java.io.*;

class Demo2 {

  public static FileInputStream f1(String fileName)
  {
    FileInputStream fis = null;
    try
    {
      fis = new FileInputStream(fileName);
    }
    catch (FileNotFoundException ex)
    {
      System.out.println("f1: Oops, FileNotFoundException caught");
      throw new Error("f1: File not found");
    }
    System.out.println("f1: File input stream created");
    return fis;
  }
```

```
public static FileInputStream f2(String fileName)
{
 FileInputStream fis = null;
 try
 {
  fis = new FileInputStream(fileName);
 }
 catch (FileNotFoundException ex)
 {
  System.out.println("f2: Oops, FileNotFoundException caught");
  return fis;
 }
 finally
 {
  System.out.println("f2: finally block");
  return fis;
 }

 // Compiler error: statement not reacheable
 // System.out.println("f2: Returning from f2");
 // return fis;
}

public static void main(String args[])
{
 FileInputStream fis1 = null;
 FileInputStream fis2 = null;
 String fileName = "foo.bar";
 // String fileName = null;

 System.out.println(  "main: Starting " + Demo2.class.getName()
                  + " with file name = " + fileName);

 // get file input stream 1
 try {
  fis1 = f1(fileName);
 }
 catch (Exception ex)
 {
  System.out.println("main: Oops, general exception caught");
 }
 catch (Throwable th)
 {
```

```
      System.out.println("main: Oops, throwable object caught");
    }

  // get file input stream 2
    fis2 = f2(fileName);
    System.out.println("main: " + Demo2.class.getName() + " ended");
  }
}
}
```

EXERCISE-26

MAIN THREAD CLASS

Aim: To use Thread class in java program using Multi threading.

Learning Objectives:

- Use of Threads.

```
public class MainThreadClass {

    public static void main(String[] args) {

        ThreadClass dt1=new ThreadClass("Run");
        ThreadClass dt2=new ThreadClass("Thread");
        dt1.start(); // this will start thread of object 1
        dt2.start(); // this will start thread of object 2
    }
}

public class ThreadClass extends Thread{

    String msg;

    public void run()
    {
        for(int i=0;i<=5;i++)
        {
            System.out.println("Run method: "+msg);
        }
    }

    ThreadClass(String mg)
```

```
{
   msg=mg;    } }
```

EXERCISE-27

CREATING A NEW THREAD USING RUNNABLE INTERFACE

Aim: To create a new thread using Runnable interface.

Learning Objectives:

- Use of threads.
- Use of runnable interface.

```
// Create a new thread.

class NewThread implements Runnable {
  Thread t;
  NewThread() {
    // Create a new, second thread
    t = new Thread(this, "Demo Thread");
    System.out.println("Child thread: " + t);
    t.start(); // Start the thread
  }
 // This is the entry point for the second thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Child Thread: " + i);
        // Let the thread sleep for a while.
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
  }
}
class ThreadConcept {
  public static void main(String args[]) {
    new NewThread(); // create 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.");
  }
}
```

EXERCISE-28

USUAGE OF JAVA THREAD CLASS

Aim: To use Java Thread class.

Learning Objectives:

- Use of Java Thread Class.

```
// Create a second thread by extending Thread

class NewThread extends Thread {
  NewThread() {
    // Create a new, second thread
    super("Demo Thread");
    System.out.println("Child thread: " + this);
    start(); // Start the thread
  }
  // This is the entry point for the second thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Child Thread: " + i);
                              // Let the thread sleep for a while.
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
  }
}
class ExtendThread {
```

```
  public static void main(String args[]) {
    new NewThread(); // create 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.");
  }
}
```

EXERCISE-29

INTERPROCESS COMMUNICATION

Aim: To implement interprocess communication.

Learning Objectives:

- Use of Interprocess Communication.

```
class Shared {
int num=0;
boolean value = false;
 synchronized int get() {
 if (value==false)
 try {
   wait();
   }
 catch (InterruptedException e) {
 System.out.println("InterruptedException caught");
   }
System.out.println("consume: " + num);
value=false;
notify();
return num;
}
synchronized void put(int num) {
 if (value==true)
 try {
   wait();
   }
```

```
  catch (InterruptedException e) {
  System.out.println("InterruptedException caught");
    }
    this.num=num;
    System.out.println("Produce: " + num);
    value=false;
    notify();
    }
    }

    class Producer extends Thread {
  Shared s;
  Producer(Shared s) {
    this.s=s;
    this.start();
  }
  public void run() {
    int i=0;
    s.put(++i);
    }
}
class Consumer extends Thread{
  Shared s;
  Consumer(Shared s) {
    this.s=s;
    this.start();
  }
  public void run() {
    s.get();
    }
}
public class InterThread{
  public static void main(String[] args)
  {
    Shared s=new Shared();
    new Producer(s);
    new Consumer(s); }}
```

EXERCISE-30

THREAD GROUP

Aim: To use thread group concept.

Learning Objectives:

- Use of Thread groups.

```
public class ThreadGroupDemo {
class MyThreadGroup extends ThreadGroup {
public void uncaughtException(Thread t, Throwable ex) {
System.err.println("I caught " + ex);
}
public MyThreadGroup(String name) {
super(name);
}
}
public static void main(String[] args) {
new ThreadGroupDemo().work();
}
protected void work() {
ThreadGroup g = new MyThreadGroup("bulk threads");
Runnable r = new Runnable() {
public void run() {
System.out.println(Thread.currentThread().getName() + " started");
for (int i=0; i<5; i++) {
System.out.println(Thread.currentThread().getName() + ": " + i);
try {
Thread.sleep(1776);
} catch (InterruptedException ex) {
System.out.println("Huh?");
}
}}
```

……………………………………………………………………………

LAB SESSION-4

Lab session-4 contains the Unit-4(Exercise-31to36) topics.
Programs based on the logic, topic coverage and example Application.

EXERCISE-31

IMPORT STATEMENT

Aim: To use import statement.

Learning Objectives:

- Use of importing java package applets.

```
import java.applet.*;    // Don't forget these import statements!
import java.awt.*;
public class FirstApplet extends Applet {
   // This method displays the applet.
   // The Graphics class is how you do all drawing in Java.
   public void paint(Graphics g) {
      g.drawString("Hello World", 25, 50);
   }
```

EXERCISE-32

MOUSE EVENTS

Aim: To use mouse events with java application.

Learning Objectives:

- Use of mouse events.

```
import java.applet.*;
import java.awt.*;
public class Scribble extends Applet {
   private int last_x = 0;
   private int last_y = 0;
```

```
 // called when the user clicks
  public boolean mouseDown(Event e, int x, int y)
  {
     last_x = x; last_y = y;
     return true;
  }
  // called when the mouse moves with the button down
  public boolean mouseDrag(Event e, int x, int y)
  {
     Graphics g = getGraphics();
     g.drawLine(last_x, last_y, x, y);
     last_x = x;
     last_y = y;
     return true;
  }
}
```

EXERCISE-33

CREATION OF ANIMATION USING APPLET

Aim: To use applets for creating animation.

Learning Objectives:

- Use of applets.
- Use of animations.

```
import java.applet.*;
import java.awt.*;
import java.net.*;
import java.util.*;
Public class Animator extends Applet implements Runnable {
  protected Image[] images;
   protected int current_image;
  // Read the basename and num_images parameters.
  // Then read in the images, using the specified base name.
  // For example, if basename is images/anim, read images/anim0,
  // images/anim1, etc.  These are relative to the current document URL.

  public void init() {
     String basename = this.getParameter("basename");
     int num_images;
     try { num_images = Integer.parseInt(this.getParameter("num_images")); }
```

```
    catch (NumberFormatException e) { num_images = 0; }
   images = new Image[num_images];
    for(int i = 0; i < num_images; i++) {
      images[i] = this.getImage(this.getDocumentBase(), basename + i);
    }
  }
  // This is the thread that runs the animation, and the methods
  // that start it and stop it.
  private Thread animator_thread = null;
  public void start() {
    if (animator_thread == null) {
      animator_thread = new Thread(this);
      animator_thread.start();
    }
  }
  public void stop() {
    if ((animator_thread != null) && animator_thread.isAlive())
      animator_thread.stop();
    // We do this so the garbage collector can reclaim the Thread object.
    // Otherwise it might sit around in the Web browser for a long time.
    animator_thread = null;
  }

  // This is the body of the thread--the method that does the animation.
  public void run() {
    while(true) {
      if (++current_image >= images.length) current_image = 0;
      this.getGraphics().drawImage(images[current_image], 0, 0, this);
      this.getToolkit().sync();  // Force it to be drawn *now*.
      try { Thread.sleep(200); } catch (InterruptedException e) { ; }
    }
  }
}
```

EXERCISE-34

READING INPUT USING APPLETS

Aim: To read input using applets.

Learning Objectives:

- Use of applets.

```
A)import java.applet.*;
  import java.awt.*;

/**
 * A Java applet parameter test class.
 * Demonstrates how to read applet parameters.
 */
public class AppletParameterTest extends Applet {

  public void paint(Graphics g) {

    String myFont   = getParameter("font");
    String myString = getParameter("string");
    int mySize      = Integer.parseInt(getParameter("size"));
    Font f = new Font(myFont, Font.BOLD, mySize);
    g.setFont(f);
    g.setColor(Color.red);
    g.drawString(myString, 20, 20);
  }
}
```

b) AppletParameterTest.html.

```
<HTML>
<HEAD>
<TITLE>Java applet example - Passing applet parameters to Java
applets</TITLE>
</HEAD>
<BODY>
```

```
<APPLET CODE="AppletParameterTest.class" WIDTH="400"
HEIGHT="50">
   <PARAM NAME="font"    VALUE="Dialog">
   <PARAM NAME="size"    VALUE="24">
   <PARAM NAME="string"  VALUE="Hello, world ... it's me.   :)">
</APPLET>
</BODY>
</HTML>
<!-- AppletParameterTest.html - an HTML file that
    calls a Java applet with the applet tag,
    and passes several parameters to the applet via
    the <PARAM> tag. -->
```

EXERCISE-35

User interface development using Java SWINGS package

Aim: Developing user interface.

Learning Objectives:

- Use of swings API.

```
import javax.swing.*;
public class HelloWorldSwing {
   /**
    * Create the GUI and show it.  For thread safety,
    * this method should be invoked from the
    * event-dispatching thread.
    */
   private static void createAndShowGUI() {
      //Make sure we have nice window decorations.
      JFrame.setDefaultLookAndFeelDecorated(true);
      //Create and set up the window.
      JFrame frame = new JFrame("HelloWorldSwing");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

     //Add the ubiquitous "Hello World" label.
      JLabel label = new JLabel("Hello World");
      frame.getContentPane().add(label);

      //Display the window.
```

```
        frame.pack();
        frame.setVisible(true);
    }
    public static void main(String[] args) {
        //Schedule a job for the event-dispatching thread:
        //creating and showing this application's GUI.
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                createAndShowGUI();
            }
        }
}    }
}
```

EXERCISE-36

ADDING ITEMS AND REMOVING ITEMS FROM JLIST

Aim: To add items and remove items from list component user interface

Learning Objectives:

- Use of lists.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
    public class PhilosophersJList extends JFrame {
     private DefaultListModel philosophers;
     private JList list;
     public PhilosophersJList()
        {
        super( "Favorite Philosophers" );
       // create a DefaultListModel to store philosophers
        philosophers = new DefaultListModel();
        philosophers.addElement( "Yandamuri Veerendranath" );
        philosophers.addElement( "Yaddapudi Sulochana Rani" );
        philosophers.addElement( "Thripura Gopi Chand" );
        philosophers.addElement( "Karuna Nithi" );
        philosophers.addElement( "Thenali Rama Krishna" );
        philosophers.addElement( "thikkana" );
        philosophers.addElement( "Sri Sri" );
        philosophers.addElement( "Sri Krishna Devaraya" );

        // create a JList for philosophers DefaultListModel
```

```
  list = new JList( philosophers );
 // allow user to select only one philosopher at a time
  list.setSelectionMode(
  ListSelectionModel.SINGLE_SELECTION );
 // create JButton for adding philosophers
  JButton addButton = new JButton( "Add Philosopher" );
  addButton.addActionListener(
     new ActionListener() {
    public void actionPerformed( ActionEvent event )
        {
        // prompt user for new philosopher's name
        String name = JOptionPane.showInputDialog(
        PhilosophersJList.this, "Enter Name" );
       // add new philosopher to model
        philosophers.addElement( name );
     }
  }
  );
 // create JButton for removing selected philosopher
  JButton removeButton =   new JButton( "Remove Selected Philosopher" );

  removeButton.addActionListener(  new ActionListener() {
  public void actionPerformed( ActionEvent event )
        {
        // remove selected philosopher from model
        philosophers.removeElement(
        list.getSelectedValue() );
     }
  }
  );
     // lay out GUI components
  JPanel inputPanel = new JPanel();
  inputPanel.add( addButton );
  inputPanel.add( removeButton );
  Container container = getContentPane();
  container.add( list, BorderLayout.CENTER );
  container.add( inputPanel, BorderLayout.NORTH );
 setDefaultCloseOperation( EXIT_ON_CLOSE );
  setSize( 400, 300 );
  setVisible( true );

} // end PhilosophersJList constructor
 // execute application
public static void main( String args[] )
```

```
        {
        new PhilosophersJList();
    }
        // create JButton for removing selected philosopher
        JButton removeButton =
        new JButton( "Remove Selected Philosopher" );
        removeButton.addActionListener( new ActionListener() {
        public void actionPerformed( ActionEvent event )
                {
                // remove selected philosopher from model
                philosophers.removeElement(
                list.getSelectedValue() );
            }
        }
        );
         // lay out GUI components
        JPanel inputPanel = new JPanel();
        inputPanel.add( addButton );
        inputPanel.add( removeButton );
       Container container = getContentPane();
        container.add( list, BorderLayout.CENTER );
        container.add( inputPanel, BorderLayout.NORTH );
        setDefaultCloseOperation( EXIT_ON_CLOSE );
        setSize( 400, 300 );
        setVisible( true );
        } // end PhilosophersJList constructor
       // execute application
    public static void main( String args[] )
        {
        new PhilosophersJList();
    }
}
```

****

LAB SESSION-5

Lab session-5 covers the programs of Unit-5(Exercise-37to40) topics.
Programs based on JDBC, Sockets, URL.

EXERCISE-37

JDBC APPLICATION USING SELECT

Aim: To use JDBC for selection**.**

Learning Objectives:

- Use of JDBC.
- Use of URL.

```
import java.net.URL;
import java.sql.*;
class Select {
    public static void main(String argv[]) {
                try {
                        // Create a URL specifying an ODBC data source
name.
                        String url = "jdbc:odbc:wombat";
                // Connect to the database at that URL.
        Connection con = DriverManager.getConnection(url,"kgh", "");
                // Execute a SELECT statement
                        Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT a, b, c, d, key FROM Table1");
                // Step through the result rows.
                        System.out.println("Got results:");
                        while (rs.next()) {
                                // get the values from the current row:
                                int a = rs.getInt(1);
                                BigDecimal b = rs.getBigDecimal(2);
                                char c[] = rs.getString(3).tocharArray();
                                boolean d = rs.getBoolean(4);
                                String key = rs.getString(5);
                                // Now print out the results:
```

```
                    System.out.print("  key=" + key);
                    System.out.print("  a=" + a);
                    System.out.print("  b=" + b);
                    System.out.print("  c=");
                    for (int i = 0; i < c.length; i++) {
                        System.out.print(c[i]);
                    }
                    System.out.print("  d=" + d);
                    System.out.print("\n");
                }
                stmt.close();
                con.close();
            } catch (java.lang.Exception ex) {
                ex.printStackTrace();
            }          } }
```

EXERCISE-38

JDBC APPLICATION USING UPDATE

Aim: To use JDBC concept for updation.

Learning Objectives:

- Use of JDBC.
- Use of URL.

```
import java.net.URL;
import java.sql.*;
class Update {
        public static void main(String argv[]) {
                try {
                        // Create a URL specifying an ODBC data source
name.
                        String url = "jdbc:odbc:wombat";
                // Connect to the database at that URL.
        Connection con = DriverManager.getConnection(url, "kgh", "");
        // Create a prepared statement to update the "a" field of a
                        // row in the "Table1" table.
                        // The prepared statement takes two parameters.
                        PreparedStatement stmt = con.prepareStatement(
                        "UPDATE Table1 SET a = ? WHERE key = ?");
                        // First use the prepared statement to update
                        // the "count" row to 34.
                        stmt.setInt(1, 34);
```

```
                    stmt.setString(2, "count");
                    stmt.executeUpdate();
                    System.out.println("Updated \"count\" row OK.");
            // Now use the same prepared statement to update the
                    // "mirror" field.
        // We rebind parameter 2, but reuse the other parameter.
                    stmt.setString(2, "mirror");
                    stmt.executeUpdate();
            System.out.println("Updated \"mirror\" row OK.");
        stmt.close();
        con.close();
            } catch (java.lang.Exception ex) {
                    ex.printStackTrace();
            }
        }
}
```

EXERCISE-39

Using Java SOCKET CLASSES FOR SERVER AND CLIENT COMMUNICATIONAPPLICATION

Aim: To use sockets application for servers.

Learning Objectives:

- Use of sockets.

```
import java.io.*;
import java.net.*;
public class Provider{
        ServerSocket providerSocket;
        Socket connection = null;
        ObjectOutputStream out;
        ObjectInputStream in;
        String message;
        Provider(){}
        void run()
        {
            try{
                    //1. creating a server socket
                    providerSocket = new ServerSocket(2004, 10);
                    //2. Wait for connection
                    System.out.println("Waiting for connection");
```

```
			connection = providerSocket.accept();
System.out.println("Connection received from " +
connection.getInetAddress().getHostName());
			//3. get Input and Output streams
			out = new
ObjectOutputStream(connection.getOutputStream());
			out.flush();
			in = new
ObjectInputStream(connection.getInputStream());
			sendMessage("Connection successful");
	//4. The two parts communicate via the input and output streams
			do{
				try{
					message = (String)in.readObject();
				System.out.println("client>" + message);
					if (message.equals("bye"))
						sendMessage("bye");
				}
				catch(ClassNotFoundException classnot){
			System.err.println("Data received in unknown format");
				}
			}while(!message.equals("bye"));
		}
		catch(IOException ioException){
			ioException.printStackTrace();
		}
		finally{
			//4: Closing connection
			try{
				in.close();
				out.close();
				providerSocket.close();
			}
			catch(IOException ioException){
				ioException.printStackTrace();
			}
		}
	}
	void sendMessage(String msg)
	{
		try{
			out.writeObject(msg);
			out.flush();
			System.out.println("server>" + msg);
```

```
                }
                catch(IOException ioException){
                        ioException.printStackTrace();
                }
        }
        public static void main(String args[])
        {
                Provider server = new Provider();
                while(true){
                        server.run();
                }
        }
}
```

EXERCISE-40

SOCKETS APPLICATION FOR CLIENT

Aim: To use sockets application for client.

Learning Objectives:

- Use of sockets.

```
import java.io.*;
import java.net.*;
public class Requester{
        Socket requestSocket;
        ObjectOutputStream out;
        ObjectInputStream in;
        String message;
        Requester(){}
        void run()
        {
                try{
                        //1. creating a socket to connect to the server
                        requestSocket = new Socket("localhost", 2004);
                System.out.println("Connected to localhost in port 2004");
                        //2. get Input and Output streams
                        out = new
ObjectOutputStream(requestSocket.getOutputStream());
                        out.flush();
                        in = new
ObjectInputStream(requestSocket.getInputStream());
                        //3: Communicating with the server
```

```
			do{
				try{
					message = (String)in.readObject();
				System.out.println("server>" + message);
					sendMessage("Hi my server");
					message = "bye";
					sendMessage(message);
				}
				catch(ClassNotFoundException classNot){
			System.err.println("data received in unknown format");
				}
			}while(!message.equals("bye"));
		}
		catch(UnknownHostException unknownHost){
	System.err.println("You are trying to connect to an unknown host!");
		}
		catch(IOException ioException){
			ioException.printStackTrace();
		}
		finally{
			//4: Closing connection
			try{
				in.close();
				out.close();
				requestSocket.close();
			}
			catch(IOException ioException){
				ioException.printStackTrace();
			}
		}
	}
	void sendMessage(String msg)
	{
		try{
			out.writeObject (msg);
			out.flush();
			System.out.println("client>" + msg);
		}
		catch(IOException ioException){
			ioException.printStackTrace();
		}
	}
	public static void main(String args[])
	{
```

```
Requester client = new Requester();
client.run(); }}
```

****