INTRODUCTION TO INTERNET PROGRAMMING (JAVA)

C-126

Self Learning Material

Directorate of Distance Education

SWAMI VIVEKANAND SUBHARTI UNIVERSITY
MEERUT-250005
UTTAR PRADESH

SIM Module Developed by : Raman Chadha

Reviewed by the Study Material Assessment Committee Comprising:

1. **Lt. (Gen.) B.S. Rathore, Vice-Chancellor**
2. **Dr. Sushmita Saxena, Pro-Vice-Chancellor**
3. **Dr. Mohan Gupta**
4. **Mr. Shashiraj Teotia**
5. **Mr. Ashish Bhatnagar**

Edition: 2017

Published by : Laxmi Publications Pvt Ltd., 113, Golden House, Daryaganj, New Delhi-110 002.
Tel: 43532500, E-mail: info@laxmipublications.com

DEM-2231-86.50-INTERNET PROG (JAVA) C-126
Typeset at: Goswami Associates, Delhi

Printed at:

CONTENTS

Units		Page No.
I.	Fundamentals of Java Programming	1
II.	Data Types, Operators and Arrays	13
III.	Classes and Objects in Java	35
IV.	Exception Handling	56
V.	Packages and Interfaces	72
VI.	Multithreaded Programming	85
VII.	I/O In Java	108
VIII.	Applets	132
IX.	Graphics and User Interface	153

SYLLABUS

INTRODUCTION TO INTERNET PROGRAMMING (JAVA) (BCA)
C-126

FUNDAMENTALS OF JAVA PROGRAMMING

Unit 1: Introduction to Java

1.1 Applets and Applications
1.2 Java Buzzwords
1.3 The Java Plateform
1.4 Java Libraries
1.5 Starting With Java

Unit 2: Data Types, Operators and Arrays

2.1 Data Types In Java
2.2 Operators
2.3 Java Keywords
2.4 Mixing Datatypes
2.5 Type Casting
2.6 Programming Constructors in Java
2.7 Arrays

Unit 3: Classes and Objects In Java

3.1 Classes and Objects
3.2 Constructor
3.3 Subclassing
3.4 The Extends Keyword
3.5 The instance of Operator
3.6 Static Variables and Methods
3.7 The Final Keyword
3.8 Access Control
3.9 Method Overriding
3.10 Abstract Classes
3.11 Inner Classes

Unit 4: Exception Handling

4.1 Exception Classes
4.2 Using Try and Catch
4.3 Handling Multiple Exceptions
4.4 Sequencing Catch Blocks
4.5 Using Finally
4.6 Built –in Exception
4.7 Throwing Exceptions
4.8 Catching Exceptions
4.9 User Defined Exception

Unit 5: Packages and Interfaces

5.1 Creating Packages
5.2 Adding Classes to Existing Packages

5.3 Interface
5.4 Creating Interfaces
5.5 Exceptions

ADVANCED CONCEPTS

Unit 6: Multithreaded Programming
6.1 Multithreading: an introduction
6.2 The Main Thread
6.3 Java Thread Model
6.4 Thread Priorities
6.5 Synchronization in Java
6.6 Interthread Communication

Unit 7: I/O in Java
7.1 I/O Basics
7.2 Streams and Stream Classes
7.2.1 Byte Stream Classes
7.2.2 Character Stream Classes
7.3 The Predefined Streams
7.4 Reading form, and Writing to, Console
7.5 Reading and Writing File
7.6 The Transient and Volatile Modifiers
7.7 Using Native Methods

Unit 8: Applets
8.1 The Applet Class
8.2 Applet Architecture
8.3 An Applet Skeleton: Initialization and Termination
8.4 Handling Events
8.5 HTML Applet Tag

Unit 9: Graphics and User Interface
9.1 Graphics Contexts and Graphics Objects
9.1.1 Color Control
9.1.2 Fonts
9.1.3 Coordinate System
9.2 User Interface Components
9.3 Building User Interface with AWT
9.4 Swing – Based GUI
9.5 Layouts and Layout Manager
9.6 Container

U N I T

1

FUNDAMENTALS OF JAVA PROGRAMMING

STRUCTURE

1.0 Learning Objectives
1.1 Applets and Applications
1.2 Java Buzzwords
1.3 The Java Platform
1.4 Java Libraries
1.5 Starting with Java
- *Summary*
- *Review Questions*

1.0. LEARNING OBJECTIVES

After going through this unit, you will be able to :

- discuss about the applets and applications
- define the java virtual machine
- explain the java buzzwords
- describe the java libraries
- differentiate the starting with java and java platform.

1.1. APPLETS AND APPLICATIONS

Now a day, Java is widely used for applications and applets. The code for the application in object format resides on the user's machine and is executed by a run-time interpreter. **Applications** are stand alone and are executed using a Java interpreter. Whereas, A **Java applet** produces object code which restricts local user resource access and can be normally be interpreted within the user's browser. Applets can be very useful, user friendly programs or can be a lot of fun. Applets are written in HTML documents and are executed from within a Java empowered **browser** such as Inter Explorer or Netscape Navigator. For instance, whenever we pull up any web page, the Java code is embedded within HTML code. On the contrary any program which is not embedded within HTML code is pertained to as an application. An Applet

NOTES

is a program that runs within a web browser. The main idea of applets is to allow the program to be launched by any Java Enabled web browser. On the other hand Java applications runs stand alone ***(can run like any program i.e. on our computer)***. It is important to note at this stage that by default, Java applets have some restrictions when it comes to dealing with the operating system. If we developed a program that would be independent from the system, then applets would be a good solution. There are certain methods which can help us to modifying those security restrictions. In order to run an application or an applet on any machine we are required to have a **JVM (Java virtual machine)** installed on our machine. Following are some important discussions and differences between applets and applications:

- Applets are the small programs while applications are larger programs.
- Applets don't have the main method while in an application execution starts with the main method. Applets can run in our browser's window or in an appletviewer.
- To run the applet in an appletviewer will be an advantage for debugging.
- Applets are the powerful tools because it covers half of the java language picture.
- Java applets are the best way of creating the programs in java.
- Both (Applets and the java applications) have the same importance at their own places. Applications are also the platform independent as well as byte oriented just like the applets.
- Applets are designed just for handling the client site problems, while the java applications are designed to work with the client as well as server.
- Applications are designed to exist in a secure area, while the applets are typically used.
- Applications and applets have much of the similarity such as both have most of the same features and share the same resources.
- Applets are created by extending the java.applet. Applet class while the java applications start execution from the main method.
- Applications are not too small to embed into a HTML page so that the user can view the application in our browser. On the other hand applets have the accessibility criteria of the resources.

The key feature is that while they have so many differences but both can perform the same purpose.

To create an applet just creates a class that extends the java.applet. Applet class and inherit all the features available in the parent class. The following programs make all the things clear:

```
import java.awt.*;
import java.applet.*;
class Myclass extends Applet{
public void init(){
/* All the variables, methods and images initialize here will
be called only once because this method is called only once when
the applet is first initializes*/
```

```
}
public void start(){
/* The components needed to be initialize more than once in
applet, are written here. This method can be called more than
once.*/
}
public void stop(){
/* This method is the counterpart to start(). The code, used to
stop the execution is written here*/
}
public void destroy(){
/* This method contains the code that result in to release the
resources to the applet before it is finished. This method is
called only once.*/
}
public void paint(Graphics g){
/* Write the code in this method to draw, write, or color things
on the applet pane are */
}}
```

In the above applet we have seen that there are five methods. In which two **(init() and destroy)** are called only once while remaining three **(start(), stop(), and paint())** can be called any number of times as per the requirements. The major difference between the two **(applet and application)** is that java applications are designed to work under the homogenous and more secure areas. Java applets are designed to run the heterogeneous and probably unsecured environment. Internet has imposed several restrictions on it. The conclusion is that the java applets provides a wide variety of formats for program execution and a very tight security model on the open environment as on the Internet.

Java applications have the majority of differences with the java applets. **Applications** are stand-alone programs that do not require the use of a browser. Java applications run by starting the Java interpreter from the command line and by specifying the file that contains the compiled application. Applications usually reside on the system on which they are deployed. Applications access resources on the system, and are restricted by the Java security model. The following program illustrates the structure of the java application:

```
public class MyClass{
/* Various methods and variable used by the class MyClass are
written here */
class myClass{
/* This contains the body of the class myClass*/
}
```

```
public static void main(String args[]){
/* The application starts it's actual execution from this place.
**/
}}
```

The main method here is nothing but the system method used to invoke the application. The code that results an action should locate in the main method. Therefore this method is more than the other method in any java application.

1.2. JAVA BUZZWORDS

Buzzwords of java are actually representation java language features as like Simple, Object Oriented, Secure and Distributed etc. No discussion of the origins of java is complete without a look at the java buzzwords. The key considerations were summed up by the java team in the following list of buzzwords:

- Simple
- Secure
- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance
- Distributed
- Dynamic.

Let's examine of this what it implies:

• **Simple:** Java was designed to be easy for the professional programmer to learn and use effectively. If you already understand the basic concepts of object-oriented programming, learning java will be even easier. Best of all, if you are an experienced C++ programmer, moving to java will require very little effort. Because java inherits the C/C++ syntax and many of the object-oriented features of C++.

• **Secure:** Java is intended for use in networked/distributed environments. Toward that end, a lot of emphasis has been placed on security. *Java enables the construction of virus-free, tamper-free systems. The authentication techniques are based on public-key encryption. There is a strong interplay between "robust" and "secure."* For example, the changes to the semantics of pointers make it impossible for applications to forge access to data structures or to access private data in objects that they do not have access to. This closes the door on most activities of viruses.

• **Portable:** Being architecture neutral is a big chunk of being portable, but there's more to it than that. Unlike C and C++, there are no "implementation dependent" aspects of the specification. *The sizes of the primitive data types are specified, as is the behavior of arithmetic on them.* For example, "int" always means a signed two's complement 32 bit integer, and "float" always means a 32-bit IEEE 754 floating point

number. Making these choices is feasible in this day and age because essentially all interesting CPUs share these characteristics.

• **Object-Oriented:** Although influenced by its predecessors, java was not designed to be source-code compatible with any other language. The object model in java is simple and easy to extend, while simple types, such as integers, are kept as high performance non-objects. But object-oriented design is very powerful because it facilitates the clean definition of interfaces and makes it possible to provide reusable "software ICs." Simply stated, *object-oriented design is a technique that focuses design on the data (= objects) and on the interfaces to it.*

• **Robust:** The multi platformed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of java. To gain reliability, java restricts you in a few key areas, to force you to find your mistakes early in program development. At the same time, java frees you from having to worry about many of the most common causes of programming errors. Because java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time. To better understand how java is robust, consider tow of the main reasons for program.

Web 2.0 is the latest buzzwords. Web 2.0 is the business revolution in the computer industry caused by the move to the internet as platform, and an attempt to understand the rules for success on that new platform.

• **Multithreaded:** There are many things going on at the same time in the world around us. *Multithreading is a way of building applications with multiple threads* Unfortunately, writing programs that deal with many things happening at once can be much more difficult than writing in the conventional single-threaded C and C++ style. *Java has a sophisticated set of synchronization primitives that are based on the widely used monitor and condition variable paradigm.*

• **Architecture-neutral:** Java was designed to support applications on networks. In general, networks are composed of a variety of systems with a variety of CPU and operating system architectures. To enable a Java application to execute anywhere on the network, *the compiler generates an architecture-neutral object file format—the compiled code is executable on many processors, given the presence of the Java runtime system.*

This is useful not only for networks but also for single system software distribution. In the present personal computer market, application writers have to produce versions of their application that are compatible with the IBM PC and with the Apple Macintosh.

• **Interpreted:** Java bytecodes are translated on the fly to native machine instructions (interpreted) and not stored anywhere. And since linking is a more incremental and lightweight process, the development process can be much more rapid and exploratory. As a part of the bytecode stream, more compile-time information is carried over and available at runtime. This is what the linker's type checks are based on. It also makes programs more amenable to debugging.

• **High performance:** While the performance of interpreted bytecodes is usually more than adequate, there are situations where higher performance is required. The bytecodes can be translated on the fly (at runtime) into machine code for the particular CPU the application is running on. *The bytecode format was designed with generating machine codes in mind, so the actual process of generating machine code is generally*

NOTES

simple. Efficient code is produced: the compiler does automatic register allocation and some optimization when it produces the bytecodes.

• **Dynamic:** In a number of ways, Java is a more dynamic language than C or C++. It was designed to adapt to an evolving environment. For example, one major problem with C++ in a production environment is a side-effect of the way that code is implemented. If company A produces a class library (a library of plug and play components) and company B buys it and uses it in their product, then if A changes its library and distributes a new release, B will almost certainly have to recompile and redistribute their own software. In an environment where the end user gets A and B's software independently (say A is an OS vendor and B is an application vendor) problems can result. For example, if A distributes an upgrade to its libraries, then all of the software from B will break. It is possible to avoid this problem in C++, but it is extraordinarily difficult and it effectively means not using any of the language's OO features directly.

1.3. THE JAVA PLATFORM

Java (with a capital J) is a platform for application development. A platform is a loosely defined computer industry buzzword that typically means some combination of hardware and system software that will mostly run all the same software. For instance PowerMacs running Mac OS 9.2 would be one platform. DEC Alphas running Windows NT would be another.

Java solves the problem of platform-independence by using byte code. The Java compiler does not produce native executable code for a particular machine like a C compiler would. Instead it produces a special format called *byte code*. Java byte code written in hexadecimal, byte by byte, looks like this:

```
CA FE BA BE 00 03 00 2D 00 3E 08 00 3B 08 00 01 08 00 20 08
```

This looks a lot like machine language, but unlike machine language Java byte code is exactly the same on every platform. Java programs that have been compiled into byte code still need an interpreter to execute them on any given platform. The interpreter reads the byte code and translates it into the native language of the host machine on the fly. The most common such interpreter is Sun's program java (with a little j). Since, the byte code is completely platform independent, only the interpreter and a few native libraries need to be ported to get Java to run on a new computer or operating system. The rest of the runtime environment including the compiler and most of the class libraries are written in Java. All these pieces, the **javac** compiler, the java interpreter, the Java programming language, and more are collectively referred to as Java.

Table 1.1 Summarizes the key packages of the Java platform

Package	*Description*
java.beans	The JavaBeans component model for reusable, embeddable software components.
java.beans.beancontext	Additional classes that define bean context objects that hold and provide services to the JavaBeans objects they contain.

(Contd.)

java.io	Classes and interfaces for input and output. Although some of the classes in this package are for working directly with files, most are for working with streams of bytes or characters.
java.lang	The core classes of the language, such as **String**, **Math**, **System**, **Thread**, and **Exception**.
java.lang.ref	Classes that define weak references to objects. A weak reference is one that does not prevent the referent object from being garbage-collected.
java.lang.reflect	Classes and interfaces that allow Java programs to reflect on themselves by examining the constructors, methods, and fields of classes.
java.math	A small package that contains classes for arbitrary-precision integer and floating-point arithmetic.
java.net	Classes and interfaces for networking with other systems.
java.security	Classes and interfaces for access control and authentication. Supports cryptographic message digests and digital signatures.
java.security.acl	A package that supports access control lists. Deprecated and unused as of Java 1.2.
java.security.cert	Classes and interfaces for working with public key certificates.
java.security.interfaces	Interfaces used with DSA and RSA public-key encryption.
java.security.spec	Classes and interfaces for transparent representations of keys and parameters used in public-key cryptography.
java.text	Classes and interfaces for working with text in internationalized applications.
java.util	Various utility classes, including the powerful collections framework for working with collections of objects.
java.util.jar	Classes for reading and writing JAR files.
java.util.zip	Classes for reading and writing ZIP files.
javax.crypto	Classes and interfaces for encryption and decryption of data.
javax.crypto.interfaces	Interfaces that represent the Diffie-Hellman public/private keys used in the Diffie-Hellman key agreement protocol.
javax.crypto.spec	Classes that define transparent representations of keys and parameters used in cryptography.

Above Table 1.1 does not list all the packages in the Java platform. Java also defines numerous packages for graphics and graphical user interface programming and for distributed, or enterprise, computing. The graphics and GUI packages are **java.awt** and **javax.swing** and their many subpackages. The enterprise packages of Java include **java.rmi**, **java.sql**, **javax.jndi**, **org.omg.CORBA**, **org.omg.CosNaming**, and all of their subpackages.

NOTES

1.4. JAVA LIBRARIES

A library is a reusable software component that saves developers time by providing access to the code that performs a programming task. Libraries exist to assist with many different types of tasks. Library design is difficult. Most modern languages try to help the programmer create good libraries, and the Java language is no exception. Our objective is to learn how the Java language can help to build an effective library. To achieve this goal, we'll discuss the design of a simple library that facilitates implementing network servers. When creating a network server, there are a number of issues to consider:

- Listening on a socket.
- Accepting connections.
- Getting access to the streams represented by a connection.
- Processing the incoming data in some way, and sending back a response.

The **Java Library** is a set of dynamically loadable libraries that Java applications can call at runtime. Because the Java Platform is not dependent on any specific operating system, applications cannot rely on any of the existing libraries. The Java class libraries serve three purposes within the Java Platform:

- Like other standard code libraries, they provide the programmer a well-known set of useful facilities, such as container classes and regular expressions.
- In addition, the class libraries provide an abstract interface to tasks that would normally depend heavily on the hardware and operating system.
- Finally, some underlying platforms may not support all of the features a Java application expects. In these cases, the class libraries can either emulate those features using whatever is available.

The Java Class Library is almost entirely written in Java itself, except for the parts that need direct access to the hardware and operating system (such as for I/O, or bitmap graphics). The classes that give access to these functions commonly use native interface wrappers to access the API of the operating system. Almost all of the Java Class Library is stored in a single Java archive file called **"rt.jar"**, which is provided with **JRE** and **JDK** distributions. The Java Class Library **(rt.jar)** is located in the default bootstrap classpath, and does not have to be found in the classpath declared for the application.

Features of the Class Library are accessed through classes grouped by packages.

- **Java.lang** contains fundamental classes and interfaces closely tied to the language and runtime system.
- **I/O and networking:** access to the platform file system and more generally to networks, is provided through the java.io, and java.net packages.
- **Mathematics package:** java.math provides regular mathematical expressions, as well as arbitrary-precision decimals and integer's numbers.
- **Collections and Utilities:** provide built-in Collection data structures, and various utility classes, for Regular expressions, Concurrency, logging and Data compression.

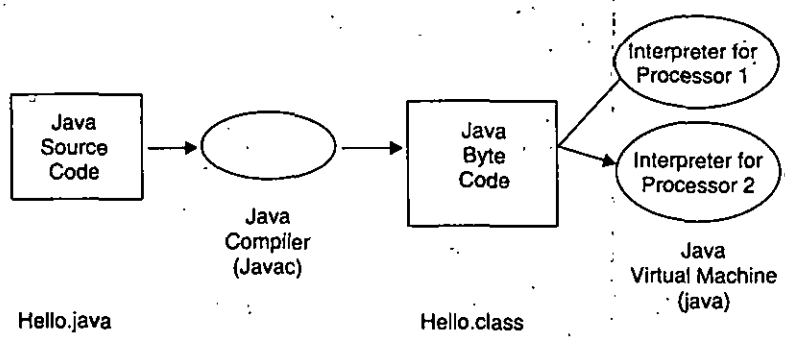
- **GUI and 2D Graphics:** the java.awt package supports basic GUI operations and binds to the underlying native system. It also contains the 2D Graphics API.
- **Sound:** provides interfaces and classes for reading, writing, sequencing, and synthesizing of sound data.
- **Text:** the java.text package deals with text, dates, numbers, and messages.
- **Image package:** java.awt.image and javax.imageio provide APIs to write, read, and modify images.
- **XML:** built-in classes handle SAX, DOM, StAX, XSLT transforms, XPath, and various APIs for Web services, as SOAP protocol and JAX-WS.
- **Databases:** access to SQL databases is provided through the java.sql package.
- **Access to Scripting engines:** the javax.script package gives access any Scripting language that conforms to this API.
- **Applets:** java.applet allows applications to be downloaded over a network and run within a guarded sandbox.
- **Java Beans:** java.beans provides ways to manipulate reusable components.

1.5. STARTING WITH JAVA

Many computers already have Java preinstalled. To find out, open a command prompt and type java. If an error receives, then the program is unknown and we have to need to install Java. To compile Java programs, we required the Java Development Kit (JDK). This contains compiler and other necessary tools. To run Java programs, you will need the Java Runtime Environment (JRE).

Writing Hello word: Most programmers start out with a Hello World program; this is the most basic program in any language. Java developed by Sun Microsystems; first release in 1995.

- Java is a first programming language to make it easy to write programs that can be executed on the Web.
- An **Object-oriented** language.
- An **platform-independent** language—by the use of **byte-code** and the **Java Virtual Machine (JVM) interpreter**.



```
/* filename: Hello.java
A Simple application program which displays "Hello, world!" to
the terminal. */

public class Hello
{public static void main(String[ ] args)
 {System.out.println("Hello, world!");// message string in ".."
in one line
 }}
```

- **Every Java program is a class.**
- This program MUST be written in a file called **Hello.java** – because the class name is Hello.
- Java is **case sensitive** (**Hello** is different from **hello**).
- The "**main**" in Java is ALWAYS indicated by "**public static void main(String[] args)**"–where "**args**" is a variable chosen by the user.
- Every **statement** in Java ends with a semicolon;
- A multi-line comment is bounded by /* */, and a ingle-line comment starts with//.
- **System.out** is the console terminal.
- **println** is a "method" which prints a string in one line (followed by a newline/carriage return).

To compile Java code, we need to use the 'javac' tool. From a command line, the command to compile this program is: **javac Hello.java**

For this to work, the javac must be in shell's path or must explicitly specify the path to the program. If the compilation is successful, javac will quietly end and return to a command prompt. To run the program simply you can run with the command: **java Hello** and then see the result or output like: **Hello, world !**

- **Classes:** In Java, everything is an object, because Java is an object oriented language. In object oriented programming we use what are called classes. Class is short for classification, and can be thought of as meaning "thing" or "item" or similar terms. A class has attributes and functions just like real objects. On my desk there is a water bottle. It has attributes like dimensions, color, volume, and water level. All water bottles have these attributes. In Java (and other programming languages), these are modeled as variables.
- **Instances:** The distinction between an object and a class is tricky but important. A class can be thought of as a type, category, or definition. For example, the class "water bottle" consists of all the characteristics and functions that are common to all water bottles. An object, however, is an *instance* of a class. The particular water bottle on the desk is an object because it is an instance of a more general class. For functionality, my water bottle has abilities like drinking water and throwing it at someone. In Object Oriented programming these are called methods. Methods are things that can do to or with an Object. The public keyword indicates visibility. It indicates that everyone can

see this class. In Java these are a few visibility levels, but for completeness they are:

- Public - Everyone can see it regardless of that Object's relationship to this one.
- Private - Only this class can see these items, nothing outside can, not even inheriting classes.
- Protected - These items can only be seen within this class or its inheriting classes.
- Default- If no visibility is specified, then only classes in the same package can see this item.

Here in this program, we have one class called **Hello** and it has no attributes. It does have one method, called main.

- **Main Method:** Now main is a very special method or function in Java. Main is where a program begins. In Java, every program must have a main method declared just like the one we see above. From above we know that it is public so that it will be visible to all classes, main methods in Java must be declared in this way.
- **Static:** The **static** keyword means simply that this method exists without the need for an instance of an object of this class. Function, main is the first method to run, if main could not be run without an instance of Hello, then how would we get an instance of Hello? We would not be able to run anything in that case.
- **Return type:** The **return type** of main is void, which simply means it returns nothing. When main finishes, the program simply ends so no return is required. Some prefer to use int main and return a value of 0 on exit to indicate successful execution.
- **Parameters:** The main function takes one parameter called args. Args is an array of type String which is indicated by the line **String[]**. Args is used to receive command line arguments if this program is called from the command line. Each argument would end up in a different cell for the array.
- **Braces:** In Java, braces {} are used to begin and end blocks. We see one opening brace {after the class declaration, which signifies the beginning of the class's contents. We next see an opening brace {on the line declaring main to indicate the beginning of its contents. We then see a} after the System.out.println statement, which closes the main function and ends its contents. Braces are always paired for each opening brace { there must be a closing brace}. Finally, the class is closed with the last closing brace}.
- **Printing:** The statement in the function that says System.out.println is a print statement. It sends output to the Java console window. It is actually a call to a built in Java function that takes a parameter of a String to be printed. Here we supply the String **"Hello World"** as a literal String. Strings in Java are always enclosed in quotations marks "".
- **Semicolon:** Finally, this line ends in a semicolon(;). In Java, semicolons end all lines of the program other than control constructs like for, if and while or function and class declarations. Everything before the semicolon is one line of code, even if it spans many lines of text.

NOTES

- **Comments:** In Java comments being with either // if they are one line or /* if they span multiple lines. Single line comments end at a carriage return and multi-line comments end at a */. Comments are ignored by the compiler and used by humans to understand their own and other people's code.
- **Saving a file:** Now, open favorite text editor and paste in this source code. Now save the file as **Hello.java** it is important that this name be exact. In Java, the source file name must have the name of the class with a .java extension, otherwise it will not compile.

SUMMARY

- Now a day, Java is widely used for applications and applets. The code for the application in object format resides on the user's machine and is executed by a run-time interpreter.
- Java enables the construction of virus-free, tamper-free systems. The authentication techniques are based on public-key encryption. There is a strong interplay between "robust" and "secure."
- The sizes of the primitive data types are specified, as is the behavior of arithmetic on them.
- The multi platformed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems.
- While the performance of interpreted bytecodes is usually more than adequate, there are situations where higher performance is required.

REVIEW QUESTIONS

1. What do you mean by applet and applications in java programming?
2. Define Java Buzzwords with suitable list example.
3. Write short notes on the following:
   (*a*) Portable
   (*b*) Multithreaded
   (*c*) Object Oriented
   (*d*) Robust
   (*e*) Dynamic.
4. What do you understand by the term java platform? Summarizes the key packages of the Java platform.
5. Describe the term java library and explain its features of the Class Library are accessed through classes grouped by packages.
6. What is the task of the main method in a java program?
7. Define the term class, main method and static word in java program.
8. Write a simple program in java to print the name, address and phone number on the screen, where class name is data.

NOTES

U N I T

2

DATA TYPES, OPERATORS AND ARRAYS

STRUCTURE

2.0 Learning Objectives
2.1 Data Types in Java
2.2 Operators
2.3 Java Keywords
2.4 Mixing Data Types
2.5 Type Casting
2.6 Programming Constructors in Java
2.7 Arrays
- *Summary*
- *Review Questions*

2.0. LEARNING OBJECTIVES

After going through this unit, you will be able to :

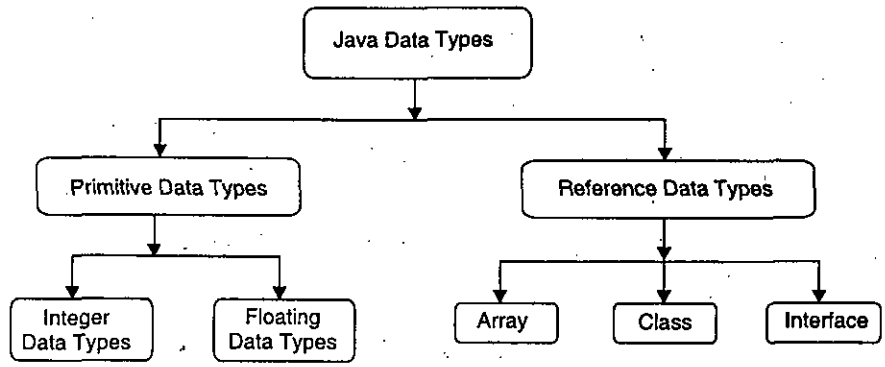
- explain data types in java
- define operators
- discuss java keywords
- explain mixing data types
- define type casting
- explain arrays.

2.1. DATA TYPES IN JAVA

Data type defines a set of permitted values on which the legal operations can be performed. In java, all the variables needs to be declared first *i.e.*, before using a particular variable, it must be declared in the program for the memory allocation process. Like: **int pedal = 1;**

This statement exists a field named **"pedal"** that holds the numerical value as **1**. The value assigned to a variable determines its **data type,** on which the legal operations

of java are performed. This behaviour specifies that, Java is a **strongly-typed** programming language. The data types in the Java programming language are divided into two categories *i.e.*, **Primitive Data Types and Reference Data Types**. Data types in java can be explained using the following hierarchy structure:



1. Primitive Data Types: The primitive data types are **predefined** data types, which always hold the value of the same data type, and the values of a primitive data type don't share the **state** with other primitive values. These data types are named by a **reserved keyword** in Java programming language. There are **eight primitive data types** supported by Java programming language:

- **Byte:** The byte data type is an 8-bit signed two's complement integer. It ranges from –128 to127 (inclusive). This type of data type is useful to save memory in large arrays. We can also use **byte** instead of **int** to increase the limit of the code. The syntax of declaring a byte type variable is shown as: **byte b = 5 ;**
- **Short:** The short data type is a 16-bit signed two's complement integer. It ranges from –32,768 to 32,767. **Short** is used to save memory in large arrays. The syntax of declaring a short type variable is shown as: **short s = 2;**
- **Int:** The int data type is used to store the integer values not the fraction values. It is a 32-bit signed two's complement integer data type. It ranges from –2,147,483,648 to 2,147,483,647 that is more enough to store large number in your program. However for wider range of values use **long**. The syntax of declaring a int type variable is shown as: **int num = 50;**
- **Long:** The long data type is a 64-bit signed two's complement integer. It ranges from –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. Use this data type with larger range of values. The syntax of declaring a long type variable is shown as: **long ln = 746;**
- **Float:** The float data type is a single-precision 32-bit IEEE 754 floating point. It ranges from 1.40129846432481707e–45 to 3.40282346638528860e+38 (positive or negative). Use a float (instead of double) to save memory in large arrays. We do not use this data type for the exact values such as currency. For that we have to use java.math.BigDecimal class. The syntax of declaring a float type variable is: **float f = 105.65 ; float f = –5000.12**
- **Double:** This data type is a double-precision 64-bit IEEE 754 floating point. It ranges from 4.94065645841246544e-324d to 1.79769313486231570e+308d (positive or negative). This data type is generally the default choice for decimal values. The syntax of declaring a double type variable is shown as:

double d = 6677.60

- **Char:** The char data type is a single 16-bit, **unsigned** Unicode character. It ranges from 0 to 65,535. They are not integral data type like int, short etc. *i.e.*, the char data type can't hold the numeric values. The syntax of declaring a char type variable is shown as: **char caps = 'c';**
- **Boolean:** The Boolean data type represents only two values: **true** and **false** and occupy is **1-bit** in the memory. These values are **keywords** in Java and represent the two boolean states: **on** or **off**, **yes** or **no.** We use **boolean** data type for specifying conditional statements as **if, while, do, for**. In Java, **true** and **false** are not the same as **True** and **False**. They are defined constants of the language. The syntax of declaring a boolean type variable is shown as: **boolean result = true;**

The ranges of these data types can be described with default values using the following table:

Data Type	*Default Value (for fields)*	*Size (in bits)*	*Minimum Range*	*Maximum Range*
byte	0	Occupy 8 bits in memory	–128	+127
short	0	Occupy 16 bits in memory	–32768	+32767
int	0	Occupy 32 bits in memory	–2147483648	+2147483647
long	0L	Occupy 64 bits in memory	–9223372036854775808	+9223372036854775807
float	0.0f	Occupy 32-bit IEEE 754 floating point	1.40129846432481707e-45	3.40282346638528860e +38
double	0.0d	Occupy 64-bit IEEE 754 floating point	4.94065645841246544 e-324d	1.79769313486231570e +308d
char	'\u0000'	Occupy 16-bit, unsigned Unicode character		0 to 65,535
boolean	false	Occupy 1- bit in memory	NA	NA

When we declare a field it is not always essential that we initialize it too. The compiler sets a default value to the fields which are not initialized which might be zero or null. However this is not recommended.

Integer Data Types: So far you would have been known about these data types. Now let's take an Integer data type in brief to better understand:

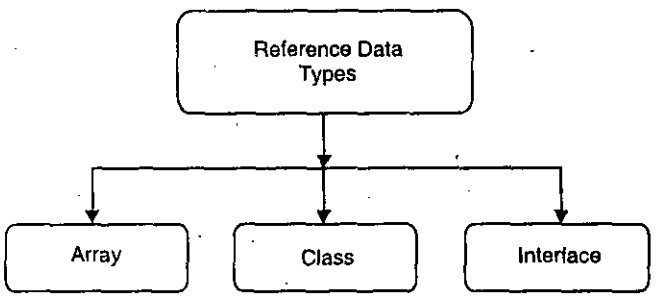
As we have described that an integer number can hold a whole number. Jav provides four different primitive integer data types that can be defined as **byte, short int,** and **long** that can store both positive and negative values. The ranges of thes data types can be described using the following table:

Data Type	*Size (in bits)*	*Minimum Range*	*Maximum Range*
byte	Occupy 8 bits in memory	–128	+127
short	Occupy 16 bits in memory	–32768	+32767
int	Occupy 32 bits in memory	–2147483648	+2147483647
long	Occupy 64 bits in memory	–9223372036854775808	+9223372036854775807

Examples of int data type are: **0, 1, 123, and 42000**

Floating-point numbers: A floating-point number represents a **real** numbe that may have a fractional values *i.e.*, In the floating type of variable, you can assig the numbers in an in a decimal or scientific notation. Floating-point numbers hav only a limited number of digits, where most values can be represented onl approximately. The *floating-point types* are **float** and **double** with a single-precisio 32-bit IEEE 754 floating point and double-precision 64-bit IEEE 754 floating poin respectively. Examples of floating-point literals are: **10.0003, 48.89 , –2000.15 , 7.04e12**

2. Reference Data Types:



In Java a **reference data type** is a variable that can contain the reference or a address of dynamically created object. These types of data type are not predefined lik **primitive** data type. The reference data types are **arrays, classes** and **interface** that are made and handle according to a programmer in a java program, which ca hold the three kinds of values as:

```
➢ array type       //Points to an array instance
➢ class type       //Points to an object or a class instance
➢ interface type   //Points to an object and a method, which is
                     implemented
                   //to the corresponding interface
```

- **Array Type:** An array is a special kind of **object** that contains values calle **elements**. The java array enables the user to store the values of the sam type in **contiguous** memory allocations. The elements in an array ar identified by an **integer index** which initially starts from **0** and ends wit **one less than number** of elements available in the array. All elements of a

NOTES

array must contain the same type of value *i.e.*, if an array is a type of integer then all the elements must be of integer type. It is a **reference data type** because the class named as **Array** implicitly extends **java.lang.Object**. The syntax of declaring the array is shown as:

```
DataType [] variable1, variable2, ..... variableN;
DataType [] variable = new DataType [ArraySize];
DataType [] variable = {item 1, item 2,...item n};
```

For example:

```
int [] a = new int [10];
String [] b = ("reference","data", "type");
```

In the first statement, an array variable "**a**" is declared of integer data type that holds the memory spaces according to the size of int. The index of the array starts from a[0] and ends with a[9]. Thus, the integer value can be assigned for each or a particular **index position** of the array.

In the second statement, the array "**b**" is declared of string data type that has the enough memory spaces to directly hold the three string values. Thus each value is assigned for each **index** position of the array.

- **class type:** We know that Java is an object-oriented programming language where an object is a variable, associated with methods that are described by a class. The name of a class is treated as a **type** in a java program, so that you can declare a variable of an object-type, and a method which can be called using that object-type variable.

Whenever a variable is created, a reference to an object is also created using the name of a class for its type *i.e.*, that variable can contain either **null** or a **reference** to an object of that class. It is not allowed to contain any other kinds of values. Such type is called **reference types** in Java. The object becomes an **instance** when the memory is allocated to that object using **new** keyword. In addition, **array types** are **reference types** because these are treated as objects in Java. For example:

```
class Fruit
 {fColor( ){....}
 fSize( ){....}
};
Fruit mango;
Fruit banana;
```

In the given example the **Fruit** is a **class** that has the reference variables as **mango and banana** through which we can call the behaviours associated with that class as **mango.fColor();** within the main method of the super class.

- **Interface Type:** Java provides an another kind of **reference data type** or a mechanism to support **multiple inheritance** feature called an **interface.** The name of an interface can be used to specify the type of a reference. A value is not allowed to be assign to a variable declared using an interface type until the **object** implements the specified **interface.**

When a class declaration implements an interface, that class inherits all of the variables and methods declared in that interface. So the implementations for all of the methods declared in the interface must be provided by that class. For example, Java provides an interface called **ActionListener** whose method named **actionPerformed()** is used to handle the different kind of event. Java also provides a class called **Thread** that implements **Runnable** interface. Thus the following assignment can be allowed:

```
Runnable r;
R = new Thread();
```

2.2. OPERATORS

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators
- Misc. Operators

- **The Arithmetic Operators:** Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Assume integer variable A holds 10 and variable B holds 20 then:

Show Examples

Operator	*Description*	*Example*
+	Addition - Adds values on either side of the operator	A + B will give 30
–	Subtraction - Subtracts right hand operand from left hand operand	A – B will give –10
*	Multiplication - Multiplies values on either side of the operator	A * B will give 200
/	Division - Divides left hand operand by right hand operand	B / A will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	B % A will give 0
++	Increment - Increase the value of operand by 1	B++ gives 21
– –	Decrement - Decrease the value of operand by 1	B– – gives 19

NOTES

- **The Relational Operators:** There are following relational operators supported by Java language

Assume variable A holds 10 and variable B holds 20 then:

Show Examples

Operator	*Description*	*Example*
==	Checks if the value of two operands is equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

- **The Bitwise Operators:** Java defines several bitwise operators which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit by bit operation. Assume if $a = 60$; and $b = 13$; Now in binary format they will be as follows:

a = 0011 1100

b = 0000 1101

$a\&b$ = 0000 1100

$a \mid b$ = 0011 1101

$a \wedge b$ = 0011 0001

$\sim a$ = 1100 0011

The following table lists the bitwise operators:

Assume integer variable A holds 60 and variable B holds 13 then:

NOTES

Show Examples

Operator	*Description*	*Example*
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -60 which is 1100 0011
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>>	Shift right zero fill operators. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

- **The Logical Operators:** The following table lists the logical operators:

Assume boolean variables A holds true and variable B holds false then:

Show Examples

Operator	*Description*	*Example*
&&	Called Logical AND operator. If both the operands are non zero then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands are non zero then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

NOTES

- **The Assignment Operators:** There are following assignment operators supported by Java language:

Show Examples

Operator	*Description*	*Example*
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+ =	Add AND assignment operator, it adds right operand to the left operand and assign the result to left operand	C + = A is equivalent to C = C + A
– =	Subtract AND assignment operator, it subtracts right operand from the left operand and assign the result to left operand	C – = A is equivalent to C = C – A
*=	Multiply AND assignment operator, it multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/ =	Divide AND assignment operator, it divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
% =	Modulus AND assignment operator, it takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<< =	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>> =	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
& =	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^ =	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
| =	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

- **Misc Operators:** There are few other operators supported by Java Language.

- **Conditional Operator (? :):** Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate boolean expressions. The goal of the operator is to decide which value should be assigned to the variable. The operator is written as :

```
variable x = (expression) ? value if true : value if false
```

Following is the example:

```
public class Test {
  public static void main(String args[]){
  int a , b;
```

```
    a = 10;
    b = (a == 1) ? 20: 30;
    System.out.println( "Value of b is :)" +  b );
    b = (a == 10) ? 20: 30;
    System.out.println( "Value of b is : " + b );
}}
```

This would produce following result:

```
Value of b is : 30
Value of b is : 20
```

• ***instance of Operator:*** This operator is used only for object reference variables. The operator checks whether the object is of a particular type(class type or interface type). instance of operator is written as:

```
( Object reference variable ) instanceOf  (class/interface type)
```

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side then the result will be true. Following is the example:

```
String name = = 'James';
boolean result = s instanceOf String;
// This will return true since name is type of String
```

This operator will still return true if the object being compared is the assignment compatible with the type on the right. Following is one more example:

```
class Vehicle {}

public class Car extends Vehicle {
   public static void main(String args[]){
      Vehicle a = new Car();
      boolean result =  a instanceof Car;
      System.out.println(result);
   }}
```

This would produce following result:

```
. true
```

• **Precedence of Java Operators:** Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example, **x = 7 + 3 * 2;** Here x is assigned 13, not 20 because operator * has higher precedence than + so it first get multiplied with 3*2 and then adds into 7. Here operators with the highest precedence appear at the top of the table; those with the

NOTES

lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	*Operator*	*Associativity*
Postfix	() [] . (dot operator)	Left to right
Unary	++ – – ! ~	Right to left
Multiplicative	* / %	Left to right
Additive	+ –	Left to right
Shift	>> >>> <<	Left to right
Relational	> >= < <=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += – = *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

2.3. JAVA KEYWORDS

Keywords are reserved words that are predefined in the language; see the table below. All the keywords are in lowercase.

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

NOTES

- **abstract:** The abstract keyword is used to declare a class or method to be abstract. An abstract method has no implementation; all classes containing abstract methods must themselves be abstract, although not all abstract classes have abstract methods. Objects of a class which is abstract cannot be instantiated, but can be extended by other classes. All subclasses of an abstract class must either provide implementations for all abstract methods, or must also be abstract.
- **boolean:** The boolean keyword is used to declare a field that can store a boolean value; that is, either true or false. This keyword is also used to declare that a method returns a value of type boolean.
- **break:** Used to resume program execution at the statement immediately following the current enclosing block or statement. If followed by a label, the program resumes execution at the statement immediately following the enclosing labeled statement or block.
- **byte:** The byte keyword is used to declare a field that can store an 8-bit signed two's complement integer. This keyword is also used to declare that a method returns a value of type byte.
- **case:** The case keyword is used to create individual cases in a switch statement; see *switch*.
- **catch:** Defines an exception handler—a group of statements that are executed if an exception is thrown in the block defined by a preceding try keyword. The code is executed only if the class of the thrown exception is assignment compatible with the exception class declared by the catch clause.
- **char:** The char keyword is used to declare a field that can store a 16-bit Unicode character. This keyword is also used to declare that a method returns a value of type char.
- **class:** A type that defines the implementation of a particular kind of object. A class definition defines instance and class fields, methods, and inner classes as well as specifying the interfaces the class implements and the immediate superclass of the class. If the superclass is not explicitly specified, the superclass is implicitly Object.
- **const:** Although reserved as a keyword in Java, const is not used and has no function.
- **continue:** Used to resume program execution at the end of the current loop body. If followed by a label, continue resumes execution at the end of the enclosing labeled loop body.
- **default:** The default can optionally be used in a switch statement to label a block of statements to be executed if no case matches the specified value.
- **do:** The do keyword is used in conjunction with while to create a do-while loop, which executes a block of statements associated with the loop and then tests a boolean expression associated with the while. If the expression evaluates to true, the block is executed again; this continues until the expression evaluates to false.
- **double:** The double keyword is used to declare a field that can hold a 64-bit double precision IEEE 754 floating-point number. This keyword is also used to declare that a method returns a value of type double.
- **else:** The else keyword is used in conjunction with if to create an if-else statement, which tests a boolean expression; if the expression evaluates to true, the block of statements associated with the if are evaluated; if it evaluates to false, the block of statements associated with the else are evaluated.

NOTES

- **enum:** A Java keyword used to declare an enumerated type. Enumerations extend the base class Enum.
- **extends:** Used in a class declaration to specify the superclass; used in an interface declaration to specify one or more super interfaces. Class X extends class Y to add functionality, either by adding fields or methods to class Y, or by overriding methods of class Y. An interface Z extends one or more interfaces by adding methods. Class X is said to be a subclass of class Y; Interface Z is said to be a sub *interface of the interfaces it extends.*
- **final:** Define an entity once that cannot be changed nor derived from later. More specifically: a final class cannot be subclassed, a final method cannot be overridden, and a final variable can occur at most once as a left-hand expression. All methods in a final class are implicitly final.
- **finally:** Used to define a block of statements for a block defined previously by the try keyword.
- **Float:** The float keyword is used to declare a field that can hold a 32-bit single precision IEEE 754 floating-point number. This keyword is also used to declare that a method returns a value of type float.
- **for:** The for keyword is used to create a for loop, which specifies a variable initialization, a boolean expression, and an incrementation. The variable initialization is performed first, and then the boolean expression is evaluated. If the expression evaluates to true, the block of statements associated with the loop are executed, and then the incrementation is performed.
- **goto:** Although reserved as a keyword in Java, goto is not used and has no function.
- **if:** The if keyword is used to create an if statement, which tests a boolean expression; if the expression evaluates to true, the block of statements associated with the if statement is executed.
- **implements:** Included in a class declaration to specify one or more interfaces that are implemented by the current class. A class inherits the types and abstract methods declared by the interfaces.
- **import:** Used at the beginning of a source file to specify classes or entire Java packages to be referred to later without including their package names in the reference.
- **instanceof:** A binary operator that takes an object reference as its first operand and a class or interface as its second operand and produces a boolean result. The instanceof operator evaluates to true if and only if the runtime type of the object is assignment compatible with the class or interface.
- **int:** The int keyword is used to declare a field that can hold a 32-bit signed two's complement integer. This keyword is also used to declare that a method returns a value of type int.
- **interface:** Used to declare a special type of class that only contains abstract methods, constant (static final) fields and static interfaces. It can later be implemented by classes that declare the interface with the implements keyword.
- **long:** The long keyword is used to declare a field that can hold a 64-bit signed two's complement integer. This keyword is also used to declare that a method returns a value of type long.

NOTES

- **native:** Used in method declarations to specify that the method is not implemented in the same Java source file, but rather in another language.
- **new:** Used to create an instance of a class or array/an object.
- **package:** A group of types. Packages are declared with the package keyword.
- **Private:** The private keyword is used in the declaration of a method, field, or inner class; private members can only be accessed by other members of their own class.
- **protected:** The protected keyword is used in the declaration of a method, field, or inner class; protected members can only be accessed by members of their own class, that class's subclasses or classes from the same package.
- **public:** The public keyword is used in the declaration of a class, method, or field; public classes, methods, and fields can be accessed by the members of any class.
- **return:** Used to finish the execution of a method. It can be followed by a value required by the method definition that is returned to the caller.
- **short:** The short keyword is used to declare a field that can hold a 16-bit signed two's complement integer. This keyword is also used to declare that a method returns a value of type short.
- **static:** Used to declare a field, method or inner class as a class field. Classes maintain one copy of class fields regardless of how many instances exist of that class. static also is used to define a method as a class method.
- **strictfp:** A Java keyword used to restrict the precision and rounding of floating point calculations to ensure portability.
- **super:** Used to access members of a class inherited by the class in which it appears. Allows a subclass to access overridden methods and hidden members of its superclass. The super keyword is also used to forward a call from a constructor to a constructor in the superclass.
- **switch:** The switch keyword is used in conjunction with case and default to create a switch statement, which evaluates a variable, matches its value to a specific case, and executes the block of statements associated with that case.
- **synchronized:** Used in the declaration of a method or code block to acquire the mutex lock for an object while the current thread executes the code. For static methods, the object locked is the class' Class. The mutex lock is automatically released when execution exits the synchronized code.
- **this:** Used to represent an instance of the class in which it appears. This can be used to access class members and as a reference to the current instance. This keyword is also used to forward a call from one constructor in a class to another constructor in the same class.
- **throw:** Causes the declared exception instance to be thrown. This causes execution to continue with the first enclosing exception handler declared by the catch keyword to handle an assignment compatible exception type.
- **throws:** Used in method declarations to specify which exceptions are not handled within the method but rather passed to the next higher level of the program.
- **transient :** Declares that an instance field is not part of the default serialized form of an object. When an object is serialized, only the values of its non-transient instance fields are included in the default serial representation.

NOTES

- **try:** Defines a block of statements that have exception handling. If an exception is thrown inside the try block, an optional catch block can handle declared exception types. A try block must have at least one catch clause or a finally block.
- **void:** The void keyword is used to declare that a method does not return any value.
- **volatile:** Used in field declarations to specify that the variable is modified asynchronously by concurrently running threads. Methods, classes and interfaces thus cannot be declared *volatile*.
- **while:** The while keyword is used to create a while loop, which tests a boolean expression and executes the block of statements associated with the loop if the expression evaluates to true; this continues until the expression evaluates to false.

Reserved words for literal values

- **false:** A boolean literal value.
- **null:** A reference literal value.
- **true:** A boolean literal value.

2.4. MIXING DATA TYPES

To learn how Java handles math operations on integers and decimals:

Integers vs. Doubles: A few starting points:

1. When integers interact with integers, the result is always integer.

2. When decimals interact with decimals, the result is always decimal.

3. When integers and decimals interact, the result will always be in decimal form.

Let's see some examples. Assume the following declarations:

```
int intt; double dub;
```

A Sampling of Mixed-Mode Arithmetic

Assignments	*Results*	*Comment*
intt = 3 + 2; dub = 3 + 2;	intt = 5 dub = 5.0	**dub** gets the decimal version of 5
intt = 3.2 + 2.5; dub = 3.2 + 2.5;	intt = **error** dub = 5.7	**intt** can't receive a decimal value - this is a loss of precision error
intt = 3.2 + 2; dub = 3.2 + 2;	intt = **error** dub = 5.2	**intt** can't receive a decimal value - this is a loss of precision error
intt = 3 + 2.5; dub = 3 + 2.5;	intt = **error** dub = 5.5	**intt** can't receive a decimal value - this is a loss of precision error
intt = 3 / 2; dub = 3 / 2;	intt = 1 dub = 1.0	**intt** and **dub** both get the truncated value of 1.5, dropping the decimal to get 1

Division with Integers: The last example requires further analysis. The rule to remember is that integers mixing with integers results in an integer. Even though the result is stored in a double, it is a truncated integer before it is stored. So how can we get the correct decimal answer when dividing integers? Below are two ways to accomplish that. Assume the variable declarations given.

```
int int1, int2; double quotient;
quotient = 1.0 * int1 / int2;
quotient = (double) int1 / int2; //NOT (double) (int1/int2);
```

The first example multiplies the numerator by a decimal, creating a decimal divided by an int, therefore resulting in a decimal.

The second example makes use of ***typecasting***, temporarily assigning or *casting* an *int* value as a *double*. Note the use of parentheses around the keyword double.

More with Typecasting: Typecasting can also enable a decimal value to be stored in an *int* variable, as a truncated integer. Recall the loss of precision errors from above when attempting to store a decimal in an *int*. If we typecast, as follows:

```
intt = (int)(3.2 + 2.5);
```

the result is that **intt** will contain **5**, the **truncated** form of **5.7**. This technique can be used in problem 12, when we want to round down to the nearest integer.

2.5. TYPE CASTING

Type casting refers to changing an entity of one datatype into another. This is important for the type conversion in developing any application. If we stored an int value into a byte variable directly, this will be illegal operation. For storing our calculated int value in a byte variable we have to change the type of resultant data. This type of operation has illustrated below:

In this example we will see that how to convert the data type by using type casting. In the given line of the code *c = (char)(t?1:0);* illustrates that if *t* which is boolean type variable is true then value of *c* which is the char type variable will be 1 but 1 is a numeric value. So, 1 is changed into character according to the Unicode value. But in this line *c = (char)(t?'1':'0');* 1 is already given as a character which will be stored as it is in the char type variable c.

Code of the program:

```
public class conversion{
  public static void main(String[] args){
    boolean t = true;
    byte b = 2;
    short s = 100;
    char c = 'C';
    int i = 200;
```

NOTES

```
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 unicode
      : " + 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);
    }}
```

2.6. PROGRAMMING CONSTRUCTORS IN JAVA

Every class has at least one it's own constructor. Constructor creates a instance for the class. Constructor initiates something related to the class's methods. Constructor is the method which name is same to the class. But there are many difference between the method *s* and the Constructor. In this example we will see that how to implement the constructor feature in a class. This program is using two classes. First class is TestConstructor and second is the main class which name is TestConstructorClient:

```
Class TestConstructor
  { int x,y;
  TestConstructor (int a, int b){
  x = a;  y = b;
}
  TestConstructor ()
  { x = 2; y = 3;
}
  int area()
  { int ar = x*y; return(ar);
}}
public class TestConstructorClient
  { public static void main(String[] args)
  { TestConstructor a = new TestConstructor ();
  System.out.println("Area of rectangle : " + a.area());
  TestConstructor b = new TestConstructor (1,1);
  System.out.println("Area of rectangle : " + b.area());
}}
```

Constructors are used to assign initial values to instance variables of the class. A default constructor with no arguments will be called automatically by the Java Virtual Machine (JVM). Constructor is always called by new operator. Constructors are declared just like as we declare methods, except that the constructor doesn't have any return type. Constructor can be overloaded provided they should have different arguments

NOTES

because JVM differentiates constructors on the basis of arguments passed in the constructor.

Whenever we assign the name of the method same as class name. Remember this method should not have any return type. This is called as constructor overloading. We have made one program on a constructor overloading, after going through it the concept of constructor overloading will get more clear. In the example below we have made three overloaded constructors each having different arguments types so that the JVM can differentiates between the various constructors.

```
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 first 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);
}}
```

2.7. ARRAYS

When you need to store same 'type' of data that can be logically grouped together, then we can go for java array. For example, imagine if we have to store the list of countries in individual java variables and manipulate them. We required more than hundred variables to manage. Array is the most important thing in any programming language. By definition, array is the static memory allocation. It allocates the memory for the same data type in sequence. It contains multiple values of same types. It also store the values in memory at the fixed size. Multiple types of arrays are used in any programming language such as:

one-dimensional, two-dimensional or can say multi-dimensional.

Declaration of an array:

```
int num[]; or int num = new int[2];
Some times user declares an array and it's size simultaneously.
We may or may not be define the size in the declaration time.
such as:
int num[] = {50,20,45,82,25,63};
```

In this program we will see how to declare and implementation. This program illustrates that the array working way. This program takes the numbers present in the num[] array in unordered list and prints numbers in ascending order. In this program the sort() function of the **java.util.*;** package is using to sort all the numbers present in the num[] array. The **Arrays.sort()** automatically sorts the list of number in ascending order by default. This function held the argument which is the array name num.

NOTES

Here is the code of the program:

```
import java.util.*;
public class array
{public static void main(String[] args)
{int num[] = {50,20,45,82,25,63};
  int l = num.length;
  int i,j,t;
  System.out.print("Given number : ");
  for (i = 0;i < l;i++ )
  {System.out.print(" " + num[i]);
    }
  System.out.println("\n");
  System.out.print("Ascending order number : ");
  Arrays.sort(num);
  for(i = 0; i < l; i++)
  {System.out.print(" " + num[i]);
 }}}
```

Output of the program:

```
C:\chandan>javac array.java
C:\chandan>java array
Given number : 50 20 45 82 25 63
Ascending order number : 20 25 45 50 63 82
```

Application of Array:

- Array is used to store elements of the **same data type.**
- Array Can be used for maintaining multiple variable names **using single name.**
- Array Can be used for **Sorting Elements**
- Array Can Perform **Matrix Operation.**
- Array Can be used in **CPU Scheduling** [Queue]
- Array Can be used in **Recursive Function**

SUMMARY

- Data type defines a set of permitted values on which the legal operations can be performed.
- The byte data type is an 8-bit signed two's complement integer. It ranges from –128 to127 (inclusive). This type of data type is useful to save memory in large arrays.
- The short data type is a 16-bit signed two's complement integer. It ranges from –32,768 to 32,767.
- The int data type is used to store the integer values not the fraction values.

NOTES

- The float data type is a single-precision 32-bit IEEE 754 floating point.
- The Boolean data type represents only two values: true and false and occupy is 1-bit in the memory.
- A floating-point number represents a real number that may have a fractional values.
- An array is a special kind of object that contains values called elements.
- Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate boolean expressions.
- Keywords are reserved words that are predefined in the language
- The boolean keyword is used to declare a field that can store a boolean value; that is, either true or false.
- The byte keyword is used to declare a field that can store an 8-bit signed two's complement integer.
- Although reserved as a keyword in Java, goto is not used and has no function.
- Used at the beginning of a source file to specify classes or entire Java packages to be referred to later without including their package names in the reference.
- Type Casting refers to changing an entity of one datatype into another. This is important for the type conversion in developing any application.

REVIEW QUESTIONS

1. Explain different types of data types in Java.
2. Differentiate between primitive data types and reference data types.
3. What are the operators used in Java programming language?
4. Write short notes on mixing Datatypes?
5. What are the type casting? Explain with suitable example.
6. How we can make a program in constructor? Explain with an example.
7. What is an array? Explain its various types.
8. What are the application of an array?

NOTES

UNIT

3

CLASSES AND OBJECTS IN JAVA

STRUCTURE

3.0 Learning Objectives
3.1 Classes and Objects
3.2 Constructor
3.3 Subclassing
3.4 The Extends Keyword
3.5 The Instanceof Operator
3.6 Static Variables and Methods
3.7 The Final Keywords
3.8 Access Control
3.9 Method Overriding
3.10 Abstract Classes
3.11 Inner Classes
- *Summary*
- *Review Questions*

3.0. LEARNING OBJECTIVES

After going through this unit, you will be able to :

- *define constructor*
- explain static variables and methods
- discuss the term access control
- explain inner classes.

NOTES

3.1. CLASSES AND OBJECTS

To understand objects, we must understand classes. To understand classes, we need to understand objects. Here we shall discuss about the concepts of Classes and Objects.

- **Object:** Objects have states and behaviors. *Example*: A dog has states-color, name, and breed as well as behaviors-wagging, barking and eating. An object is an instance of a class.
- **Class:** A class can be defined as a template/blue print that describe the behaviors/ states that object of its type support.

Objects in Java: Let us now look deep into what are objects. If we consider the real-world we can find many objects around us, Cars, Dogs, and Humans etc. All these objects have a state and behavior. If we consider a dog then its state is: name, breed, color and the behavior is; barking, wagging, running. If we compare the software object with a real world object, they have very similar characteristics. Software objects also have a state and behavior. A software object's state is stored in fields and behavior is shown via methods. So in software development methods operate on the internal state of an object and the object-to-object communication is done via methods.

A typical Java program creates many objects. Object interactions, a program can carry out various tasks, such as implementing a GUI, running an animation, or sending and receiving information over a network. Once an object has completed the work for which it was created, its resources are recycled for use by other objects. Here's a small program, called **CreateObjectDemo**, that creates three objects: one Point object and two Rectangle objects. You will need all three source files to compile this program.

```
public class CreateObjectDemo
{ public static void main(String[] args)
{   //Declare and create a point object and two rectangle
    objects.
    Point originOne = new Point(23, 94);
    Rectangle rectOne = new Rectangle(originOne, 100, 200);
    Rectangle rectTwo = new Rectangle(50, 100);

    //display rectOne's width, height, and area
    System.out.println("Width of rectOne: " + rectOne.width);
    System.out.println("Height of rectOne: " + rectOne.height);
    System.out.println("Area of rectOne: " + rectOne.getArea());

    // set rectTwo's position
    rectTwo.origin = originOne;
    //display rectTwo's position
    System.out.println("X Position of rectTwo:
    "+ rectTwo.origin.x);
```

```
    System.out.println("Y Position of rectTwo:
    "+ rectTwo.origin.y);
    //move rectTwo and display its new position
    rectTwo.move(40, 72);
    System.out.println("X Position of rectTwo:
    " + rectTwo.origin.x);
    System.out.println("Y Position of rectTwo:
    " + rectTwo.origin.y);
    }}
```

This program creates, manipulates, and displays information about various objects. Here's the output:

Width of rectOne: 100

Height of rectOne: 200

Area of rectOne: 20000

X Position of rectTwo: 23

Y Position of rectTwo: 94

X Position of rectTwo: 40

Y Position of rectTwo: 72

Classes in Java: A class is a blue print from which individual objects are created. A sample of a class is given below:

```
public class Dog
{ String breed;
  int age;
  String color;
  void barking()
  { }
  void hungry()
  { }
  void sleeping()
  { }
}
```

A class can contain any of the following variable types:

- **Local variables:** variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
- **Instance variables:** Instance variables are variables within a class but outside any method. These variables are instantiated when the class is loaded. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.

NOTES

- **Class variables:** Class variables are variables declared with in a class, outside any method, with the static keyword.

A class can have any number of methods to access the value of various kinds of methods. In the above example, barking (), hungry () and sleeping () are variables.

Below mentioned are some of the important topics that need to be discussed when looking into classes of the Java Language. The introduction to object-oriented concepts in the lesson titled Object-oriented Programming Concepts used a bicycle class as an example, with racing bikes, mountain bikes and tandem bikes as subclasses. Here is sample code for a possible implementation of a Bicycle class, to give an overview of a class declaration. Subsequent sections of this lesson will back up and explain class declarations step by step.

```
public class Bicycle
{ // the Bicycle class has three fields
  public int cadence;
  public int gear;
  public int speed;
  // the Bicycle class has one constructor
  public Bicycle(int startCadence, int startSpeed, int startGear) {
    gear = startGear;
    cadence = startCadence;
    speed = startSpeed;
  }
  // the Bicycle class has four methods
  public void setCadence(int newValue)
  { cadence = newValue;
  }
  public void setGear(int newValue)
  { gear = newValue;
  }
  public void applyBrake(int decrement)
  { speed - = decrement;
  }
  public void speedUp(int increment)
  { speed + = increment;
  }}
```

A class declaration for a MountainBike class that is a subclass of Bicycle might look like this:

```
public class MountainBike extends Bicycle
{ // the MountainBike subclass has one field
  public int seatHeight;
```

NOTES

```
// the MountainBike subclass has one constructor
public MountainBike(int startHeight, int startCadence,
int startSpeed, int startGear)
{ super(startCadence, startSpeed, startGear);
 seatHeight = startHeight;
 }
// the MountainBike subclass has one method
public void seatHeight(int newValue)
{ seatHeight = newValue;
}}
```

MountainBike inherits all the fields and methods of Bicycle and adds the field seat Height and a method to set it.

3.2. CONSTRUCTOR

A **constructor** is a special method that is used to **initialize a newly created object** and is called just after the memory is allocated for the object. It can be used to initialize the objects, to **required, or default values** at the time of object creation. It is **not mandatory** for the coder to write a constructor for the class. When discussing about classes one of the most important sub topic would be constructors. Every class has a constructor. If we do not explicitly write a constructor for a class the java compiler builds a default constructor for that class. Each time a new object is created at least one constructor will be invoked. The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor. Example of a constructor is given below:

```
class Puppy
{ public puppy()
{ }
  public puppy (String name){
  // This constructor has one parameter, name.
  }}
```

Java also supports Singleton Classes where you would be able to create only one instance of a class. Constructors are similar to methods, but with some important differences. Some of the important discussions about constructor are given below:

- **Constructor name is class name:** A constructors must have the *same name as the class its in.*
- **Default constructor:** If you don't define a constructor for a class, a *default parameterless constructor* is automatically created by the compiler. The default constructor calls the default parent constructor (super ()) and initializes all instance variables to default value (zero for numeric types, null for object references, and false for booleans).

NOTES

- **Default constructor is created only if there are no constructors:** If you define *any* constructor for your class, no default constructor is automatically created.
- **Differences between methods and constructors:**

  There is *no return type* given in a constructor signature (header). The value is this object itself so there is no need to indicate a return value.

  There is *no return statement* in the body of the constructor.

  The *first line* of a constructor must either be a call on another constructor in the same class (using this), or a call on the superclass constructor (using super). If the first line is neither of these, the compiler automatically inserts a call to the parameterless super class constructor.

If no user defined constructor is provided for a class, compiler initializes member variables to its default values.

- numeric data types are set to 0
- char data types are set to null character('\0')
- reference variables are set to null

In order to create a Constructor observe the following rules

1. It has the **same name** as the class
2. It should not return a value not even ***void***

Example:

```
class Demo
{ int value1, value2;
  Demo()
      { value1 = 10;
    value2 = 20;
    System.out.println ("Inside Constructor");
  }
  public void display(){
    System.out.println ("Value1 === " + value1);
    System.out.println ("Value2 === " + value2);
  }
  public static void main(String args[]){
    Demo d1 = new Demo ();
    d1.display ();
  }
}
```

Constructor Overloading: Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists. The compiler differentiates these constructors by taking into account the number of parameters in the list and their type. Examples of valid constructors for class Account are:

Account (int a);

Account (int a, int b);

Account (String a, int b);

Example:

```
class Demo
{ int value1, value2;
  /* Demo ()
   { value1 = 10; value2 = 20;
     System.out.println ("Inside 1st Constructor ");
   }*/
   Demo (int a){
   value1 = a;
   System.out.println ("Inside 2nd Constructor ");
   }
   Demo (int a, int b){
   value1 = a;
   value2 = b;
   System.out.println ("Inside 3rd Constructor");
   }
   public void display(){
   System.out.println ("Value1 === " + value1);
   System.out.println ("Value2 === " + value2);
   }
   public static void main(String args[]){
   Demo d1 = new Demo();
   Demo d2 = new Demo (30);
   Demo d3 = new Demo (30,40);
   d1.display();
   d2.display();
   d3.display();
}}
```

3.3. SUBCLASSING

Declaring one class to be a subclass of another (**class** ... **extends** ...)—this allows a subclass to inherit functionality from its superclass. To create an extendable class:

- all constructors, readObject, and clone must not invoke an overridable method (one that is not static, private, or final)

NOTES

- if a method depends on an implementation of a overridable method, this dependence must be explicitly stated in its javadoc.
- do not implement Serializable unless you absolutely need to.
- do not implement Cloneable unless you absolutely need to.

Consider using an "interface + default implementation" pair. This will allow users to choose the desired style of inheritance:

- subclass the default implementation directly,
- use the default implementation as a field, and forward calls to it,
- ignore the default implementation, and implement the interface entirely from scratch.

A Java subclass is a class which inherits a method or methods from a Java superclass. A Java class may be either a subclass, a superclass, both, or neither!

The Cat class in the following example is the subclass and the Animal class is the superclass.

```
public class Animal {
  public static void hide() {
    System.out.println("The hide method in Animal.");
  }
  public void override()
  { System.out.println ("The override method in Animal.");
  }}
public class Cat extends Animal
{ public static void hide() {
    System.out.println("The hide method in Cat.");
  }
  public void override()
  { System.out.println("The override method in Cat.");
  }
  public static void main(String[] args)

  { Cat myCat = new Cat();
    Animal myAnimal = (Animal)myCat;
    myAnimal.hide();
    myAnimal.override();
  }}
```

3.4. THE EXTENDS KEYWORD

The **extends** is a Java keyword, which is used in inheritance process of Java. It specifies the superclass in a class declaration using **extends** keyword. It is a keyword that indicates the parent class that a subclass is inheriting from and may not be used

as identifiers *i.e.*, you cannot declare a variable or class with this name in your Java program. In Java, every class is a subclass of **java.lang.Object**. For example, Class X extends class Y to add functionality, either by adding fields or methods to class Y, or by overriding methods of class Y. Take a look at the following example, which demonstrates the use of the 'extends' keyword.

```
public class A
{ public int number;
}
class B extends A
{ public void increment()
{ number++;
}}
```

In this example, we inherit from class A, which means that B will also contain a field called number. Two or more classes can also be inherited from the same parent class. **extends** keyword is also used in an interface declaration to specify one or more superinterfaces.

For instance:

```
interface MyInterface {
      ????.
}
interface MyInterface extends SuperInterface {
      ??????.
      }
```

3.5. THE INSTANCEOF OPERATOR

The instanceof operator can be useful to call a method based explicitly on the class of same object instead of implicitly using an overridden method and polymorphism. The instanceof operator determines if a given object is of the type of a specific class. To be more specific, the instanceof operator tests whether its first operand is an instance of its second operand. The test is made at runtime. The first operand is supposed to be the name of an object or an array element, and the second operand is supposed to be the name of a class, interface, or array type. The syntax is : **<op1> instanceof <op2>**

The result of this operation is a boolean: **true or false**. If an object specified by <op1> is an instance of a class specified by <op2>, the outcome of the operation is true, and otherwise is false. The outcome of the operation will also be true if <op2> specifies an interface that is implemented either by the class of the object specified by <op1> or by one of its superclasses. For example, consider the following code fragment:

```
interface X{}
class A implements X {}
class B extends A {}
A a = new A();
B b = new B();
```

NOTES

Note that class B does not implement the interface X directly, but class A does, and class B extends class A. Given this code, all of the following statements are true:

```
if(b instanceof X)
if(b instanceof B)
if(b instance of A)
if(a instance of A)
if(a instanceof X)
```

Knowing the type of an object at runtime is useful for the following reasons:

- Some invalid casts (explicit type conversions) involving class hierarchies cannot be caught at compile time. Therefore, they must be checked at runtime (using the instanceof operator), to avoid a runtime error.
- You might have a situation where one process is generating various kinds of objects, and the other process is processing them. The other process may need to know the object type before it can properly process it. In this situation the instanceof operator would be helpful, too.

The instanceof operator allows us to determine the type of an object. It takes an object on the left side of the operator and a type on the right side of the operator and returns a boolean value indicating whether the object belongs to that type or not. This is most clearly demonstrated with an example:

```
Package test;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;
public class InstanceofTest {
public static void main(String[] args) {
Map m = new HashMap();
System.out.println("m instanceof Map: " + (m instanceof Map));
System.out.println("m instanceof HashMap: " + (m instanceof
HashMap));
System.out.println("m instanceof Object: " + (m instanceof
Object));
System.out.println ("m instanceof Date: " + (m instanceof Date));
m=null;
System.out.println("m = null, m instanceof HashMap:
" + (m instanceof HashMap));
}}
```

If we execute the Instanceof Test class, we obtain the following results.

```
Console output from executing InstanceofTest
m instanceof Map:  true
m instanceof HashMap:  true
m instanceof Object:  true
m instanceof Date:  false
m = null, m instanceof HashMap:  false
```

We started by creating a Map object that is implemented using a HashMap. Map is an interface. Notice from the results that m is both an instanceof Map and an instanceof HashMap. Thus, we can use interfaces in addition to classes on the right side of the instanceof operator. From the results, also notice that m is an instanceof Object. This makes sense, since every object in Java inherits from Object. The instanceof operator returns true if we ask if an object is an instanceof one of its parent classes. Notice that m is not an instanceof Date, since m is a Map object and is not an implementation of the Date class.

3.6. STATIC VARIABLES AND METHODS

In place of global variables as in C/C++, Java allows variables in a class to be declared static: **public class A { static int k_var; }**

A single memory location is assigned to this variable and it exists and is accessible even when no instances of class A are created. This static variable is also called class variables, since they belong to the class as a whole. If a class property is also declared final, then it becomes a global constant (*i.e.*, can't be altered):

```
public class A { final static int k_var; }
Similarly, methods can also be declared static, and are called
class methods
public class A { static void A_method(float x){ } }
```

These class methods also can be called even when no instance of the class exists. The code in a class method can only refer to static variables and the argument variables. There are situations in which the method's behavior does not depend on the state of an object. So, there will be no use of having an object when the method itself will not be instance specific.

Variables and methods marked static belong to the class rather than to any particular instance of the class. These can be used without having any instances of that class at all. Only the class is sufficient to invoke a static method or access a static variable. A static variable is shared by all the instances of that class *i.e.*, only one copy of the static variable is maintained.

```
class Animal
{
static int animalCount=0;
public Animal()
{animalCount + =1;
} public static void main(String[] args)
{ new Animal();
  new Animal();
  new Animal();
  System.out.println("The Number of Animals is: "+ animalCount);
}}
```

The output is – ***"The Number of Animals is 3".***

A static method cannot access non-static/instance variables, because a static method is never associated with any instance. The same applies with the non-static methods as well, a static method can't directly invoke a non-static method. But static method can access non-static methods by means of declaring instances and using them.

Static members are not associated with any instances. So there is no point in using the object. So, the way static methods (or static variables) are accessed is by using the dot operator on the class name, as opposed to using it on a reference to an instance.

```
class Animal
{ static int animalCount = 0;
public Animal()
{ animalCount + =1;
}
public static int getCount()
{return animalCount;
}}
class TestAnimal
{ public static void main(String[] args)
{ new Animal();
  new Animal();
  new Animal();
  System.out.println("The Number of Animals is:
  "+ Animal.getCount());
}}
```

Remember that static methods can't be overridden. They can be redefined in a subclass, but redefining and overriding aren't the same thing. It's called as Hiding.

3.7. THE FINAL KEYWORDS

Following points must be remembers in favour of final keywords:

- A java variable can be declared using the keyword final. Then the final variable can be assigned only once.
- A variable that is declared as final and not initialized is called a blank final variable. A blank final variable forces the constructors to initialize it.
- Java classes declared as final cannot be extended. Restricting inheritance!
- Methods declared as final cannot be overridden. In methods private is equal to final, but in variables it is not.
- final parameters-values of the parameters cannot be changed after initialization. Do a small java exercise to find out the implications of final parameters in method overriding.

- Java local classes can only reference local variables and parameters that are declared as final.
- A visible advantage of declaring a java variable as static final is, the compiled java class results in faster performance.

The final keyword precedes a declared constant which after Instantiation cannot be modified.

Examples:

```
final double PI = 3.14;
PI = 1234;  // does not compile or
final double ONE;
ONE = 1;    // Instantiated
ONE = 2;    // does not compile
```

final keyword can also apply to a method or a class. When applied to a method—it means the method cannot be over-ridden. Specially useful when a method assigns a state that should not be changed in the classes that inherit it, and should use it as it is (not change the method behavior).When applied to a class it means that the class cannot be instantiated. A common use is for a class that only allows static methods as entry points or static final constants-to be accessed without a class instance.

Advantages:

- It is used only when we know that the value is not going to be changed further.
- It won't allow anyone to change that value further in anyway.
- It helps the compiler to generate code faster.

```
// final keyword code
class Another
{
final int xyz;
public Another()
{xyz=1000;
}
public void display()
{System.out.println("XYZ = " + xyz);
}}
public class FinalTest
{ static final int y = 3;
public FinalTest()
{ }
static
{ System.out.println(" Just for fun…");
}
public static void main(String[] args)
{ final int x=5;
System.out.println("X = " +x);
System.out.println("Y =" +y);
Another ano=new Another();
ano.display();
}}
```

NOTES

3.8. ACCESS CONTROL

Encapsulation links data with the code that manipulates it. However encapsulation provides another important attribute: access control. Through encapsulation, we can control what parts of a program can access the members of a class. We can define access control as the methods by which interactions with resources are limited to collections of users or programs for the purpose of enforcing integrity, confidentiality, or availability constraints. How a member can be accessed is determined by the access specifier that modifies its declaration.

Java supplies a rich set of access specifiers. Some aspects of access control are related mostly to inheritance or packages. Java's access specifiers are public, private and protected. Java also defines a default access level. When a member of a class is modified by the public specifier then that member can be accessed by any other code in your program. When a member of a class is specified as private then that member can only be accessed by other members of its class.

When no access specifier is used then by default the member of a class is public within it's own package but cannot be accessed outside of its package.

When a member of a class is specified as protected it is available to all classes in the same package and also available to all subclasses of the class that owns the protected feature. This access is provided even to subclasses that reside in a different package from the class that owns the protected feature. There are two levels of access control:

- **At the top level**—public, or *package-private* (no explicit modifier).
- At the member level—public, private, protected, or *package-private* (no explicit modifier).

A class may be declared with the modifier public, in which case that class is visible to all classes everywhere. If a class has no modifier, it is visible only within its own package. Every class has declared an access control, whether you explicitly type one or not. When a class has access to another class it means he can do one of 3 things

- Create an instance of that class.
- Extend the class (become a subclass).
- Access certain methods and variables within that class.

3.9. METHOD OVERRIDING

Below example illustrates method overriding in java. Method overriding in java means a subclass method overriding a super class method. Superclass method should be non-static. Subclass uses extends keyword to extend the super class. In the example class B is the subclass and class A is the superclass. In overriding methods of both subclass and superclass possess same signatures. Overriding is used in modifying the methods of the superclass. In overriding return types and constructor parameters of methods should match.

NOTES

Example (1):

```
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 will be displayed as:

```
C:\NewExamples>javac MethodOverriding.java

C:\NewExamples>java MethodOverriding
Sum of a and b is: 30
Sum of a, b and c is: 60
```

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden. Consider the following:

Example: (2) Method overriding.

```
class A {
int i, j;
A (int a, int b) {
i = a;
j = b;
```

```
}
// display i and j
void show()
{
System.out.println("i and j: " + i + " " + j);
}}
class B extends A
{  int k;
B(int a, int b, int c)
{ super(a, b);
  k = c;
}
// display k - this overrides show () in A
void show ()
{ System.out.println("k: " + k);
}}
Class Override {
public static void main(String args[]) {
B subOb = new B (1, 2, 3);
subOb.show(); // this calls show() in B
}}
```

The output produced by this program is shown here:

k: 3

When **show( )** is invoked on an object of type **B**, the version of **show( )** defined within **B** is used. That is, the version of **show( )** inside **B** overrides the version declared in **A**. If we wish to access the superclass version of an overridden function, then we can do so by using **super**. For example, in this version of **B**, the superclass version of **show( )** is invoked within the subclass' version.

3.10. ABSTRACT CLASSES

Java Abstract classes are used to declare common characteristics of subclasses. An abstract class cannot be instantiated. It can only be used as a superclass for other classes that extend the abstract class. Abstract classes are declared with the abstract keyword. Like any other class, an abstract class can contain fields that describe the characteristics and methods that describe the actions that a class can perform. An abstract class can include methods that contain no implementation. These are called abstract methods. The abstract method declaration must then end with a semicolon rather than a block. If a class has any abstract methods, whether declared or inherited, the entire class must be declared abstract. Abstract methods are used to provide a template for the classes that inherit the abstract methods.

NOTES

Abstract classes cannot be instantiated; they must be subclassed and actual implementations must be provided for the abstract methods. Any implementation specified can, of course, be overridden by additional subclasses. An object must have an implementation for all of its methods. You need to create a subclass that provides an implementation for the abstract method. A class abstract Vehicle might be specified as abstract to represent the general abstraction of a vehicle, as creating instances of the class would not be meaningful.

```
abstract class Vehicle {
        int numofGears;
        String color;
        abstract boolean hasDiskBrake();
        abstract int getNoofGears();
        }
```

Example of a shape class as an 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();
        }
```

We can also implement the generic shapes class as an abstract class so that we can draw lines, circles, triangles etc. All shapes have some common fields and methods, but each can, of course, add more fields and methods. The abstract class guarantees that each shape will have the same set of basic properties. We declare this class abstract because there is no such thing as a generic shape. There can only be concrete shapes such as squares, circles, triangles etc.

```
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();
        }}
```

Output:

Point: 0, 0

Notice that, in order to create a Point object, its class cannot be abstract. This means that all of the abstract methods of the Shape class must be implemented by the Point class. The subclass must define an implementation for every abstract method of the abstract superclass, or the subclass itself will also be abstract. Similarly other shape objects can be created using the generic Shape Abstract class. A big Disadvantage of using abstract classes is not able to use multiple inheritances. In the sense, when a class extends an abstract class, it can't extend any other class.

There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, 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. One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method. Here is a simple example of a class with an abstract method, followed by a class which implements that method:

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

NOTES

Notice that no objects of class **A** are declared in the program. As mentioned, it is not possible to instantiate an abstract class. One other point: class **A** implements a concrete method called **callmetoo( )**. This is perfectly acceptable. Abstract classes can include as much implementation as they see fit.

3.11. INNER CLASSES

The classes and interfaces we have seen so far in this chapter have all been top-level classes (*i.e.*, they are direct members of packages, not nested within any other classes). Starting in Java 1.1, however, there are four other types of classes, loosely known as *inner classes* that can be defined in a Java program. Used correctly, inner classes are an elegant and powerful feature of the Java language. These four types of classes are summarized here:

- **Static member classes**: A static member class is a class (or interface) defined as a static member of another class. A static method is called a class method, so, by analogy, we could call this type of inner class a "class class," but this terminology would obviously be confusing. A static member class behaves much like an ordinary top-level class, except that it can access the static members of the class that contains it. Interfaces can be defined as static members of classes.
- **Member classes** :A member class is also defined as a member of an enclosing class, but is not declared with the static modifier. This type of inner class is analogous to an instance method or field. An instance of a member class is always associated with an instance of the enclosing class, and the code of a member class has access to all the fields and methods (both static and non-static) of its enclosing class. There are several features of Java syntax that exist specifically to work with the enclosing instance of a member class. Interfaces can only be defined as static members of a class, not as non-static members.
- **Local classes**: A local class is a class defined within a block of Java code. Like a local variable, a local class is visible only within that block. Although local classes are not member classes, they are still defined within an enclosing class, so they share many of the features of member classes. Additionally, however, a local class can access any final local variables or parameters that are accessible in the scope of the block that defines the class. Interfaces cannot be defined locally.
- **Anonymous classes**: An anonymous class is a kind of local class that has no name; it combines the syntax for class definition with the syntax for object instantiation. While a local class definition is a Java statement, an anonymous class definition (and instantiation) is a Java expression, so it can appear as part of a larger expression, such as method invocation. Interfaces cannot be defined anonymously.

Use of Inner classes: Inner classes allow us to do several things.

1. Naming, if we have a class whose only purpose is for its outer class, then we can put it there to help describe its meaning.

2. non-static inner classes keep a reference to the outer class allowing it to access member variables of the outer class.
3. access, we can declare an inner class non-public to only allows private or protected access. Inner classes can also access the private variables of its outer class.

SUMMARY

- Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
- Instance variables are variables within a class but outside any method. These variables are instantiated when the class is loaded.
- Class variables are variables declared with in a class, outside any method, with the static keyword.
- A constructor is a special method that is used to initialize a newly created object and is called just after the memory is allocated for the object.
- Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists.
- The **extends** is a Java keyword, which is used in inheritance process of Java.
- A Java subclass is a class which inherits a method or methods from a Java superclass. A Java class may be either a subclass, a superclass, both, or neither!
- The instanceof operator can be useful to call a method based explicitly on the class of same object instead of implicitly using an overridden method and polymorphism. The instanceof operator determines if a given object is of the type of a specific class.
- Method overriding in java means a subclass method overriding a super class method. Superclass method should be non-static. Subclass uses extends keyword to extend the super class.
- Java Abstract classes are used to declare common characteristics of subclasses.
- Abstract classes are declared with the abstract keyword.
- A static member class is a class (or interface) defined as a static member of another class. A static method is called a class method, so, by analogy, we could call this type of inner class a "class class," but this terminology would obviously be confusing.
- A member class is also defined as a member of an enclosing class, but is not declared with the static modifier.
- A local class is a class defined within a block of Java code. Like a local variable, a local class is visible only within that block.
- An anonymous class is a kind of local class that has no name; it combines the syntax for class definition with the syntax for object instantiation.

NOTES

REVIEW QUESTIONS

1. What are the classes and objects? Explain with an example?
2. How the objects can be created in Java?
3. Write short notes on the following:
   (*i*) Constructor.
   (*ii*) Subclassing.
   (*iii*) Extends keywords.
   (*iv*) instanceof operator.
   (*v*) Final Keywords.
   (*vi*) Access Control.
4. What are the static variables and static methods in Java?
5. What is the difference between a constructor and a method?
6. Define Overriding. Explain it with an example.
7. What is abstract class? Explain it with an example?
8. What is the use of "Inner classes"? When we will use these classes?

NOTES

UNIT

4

EXCEPTION HANDLING

STRUCTURE

4.0 Learning Objectives
4.1 Exception Classes
4.2 Using Try and Catch
4.3 Handling Multiple Exceptions
4.4 Sequencing Catch Blocks
4.5 Using Finally
4.6 Built-in Exception
4.7 Throwing Exceptions
4.8 Catching Exception
4.9 User Defined Exception
• *Summary*
• *Review Questions*

4.0. LEARNING OBJECTIVES

After going through this unit, you will be able to :

- explain exception classes
- describe handing multiple exceptions
- discuss built-in exception
- define user defined exception
- explain catching exception.

4.1. EXCEPTION CLASSES

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

- A user has entered invalid data.
- A file that needs to be opened cannot be found.

NOTES

- A network connection has been lost in the middle of communications, or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

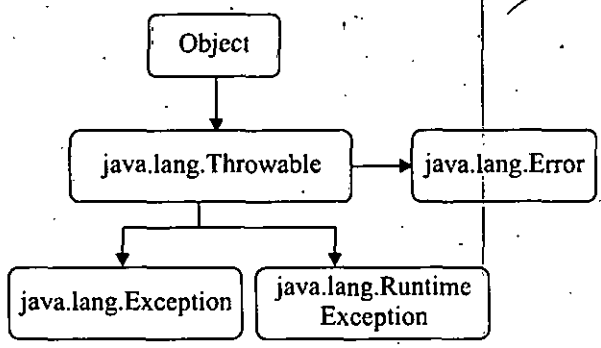
To understand how exception handling works in Java, you need to understand the three categories of exceptions:

- **Checked exceptions:** A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot simply be ignored at the time of compilation.
- **Runtime exceptions:** A runtime exception is an exception that occurs that probably could have been avoided by the programmer. As opposed to checked exceptions, runtime exceptions are ignored at the time of compilation.
- **Errors:** These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

The hierarchy of exception classes commence from **Throwable class** which is the base class for an entire family of exception classes, declared in **java.lang** package as **java.lang.Throwable.** A throwable contains a snapshot of the execution stack at the time it was created and also a message string that gives more information about the error. This class can be instantiated and thrown by the program. The throwable class is further divided into two subclasses:

1. **Exceptions:** Exceptions are thrown if any kind of unusual condition occurs that can be caught. Sometimes it also happens that the exception could not be caught and the program may get terminated. Remember that they are a member of **Exception family** and can be type of **Checked** or **Unchecked** exception.
2. **Errors:** When any kind of serious problem occurs which could not be handled easily like **OutOfMemoryError** then an error is thrown. Well, errors are not something which is thrown by you rather they are thrown by the **Java API** or by the **Java virtual machine** itself *i.e.*, only the exceptions are thrown by your code and not the errors. Also remember that they are a member of **Error family.**

The exception classes can be explained as well seeing the **exception hierarchy structure:**



NOTES

The **java.lang** package defines several classes and exceptions. Some of these classes are not checked while some other classes are checked.

Exceptions	*Description*	*Checked*	*Unchecked*
ArithmeticException	Arithmetic errors such as a divide by zero	—	YES
ArrayIndexOutOfBounds Exception	Arrays index is not within array.length	—	YES
ClassNotFoundException	Related Class not found	YES	—
IOException	InputOuput field not found	YES	—
IllegalArgumentException	Illegal argument when calling a method	—	YES
InterruptedException	One thread has been interrupted by another thread	YES	—
NoSuchMethodException	No nexistent method	YES	—
NullPointerException	Invalid use of null reference	—	YES
NumberFormatException	Invalid string for conversion to number	—	YES

As we know the exceptions are Objects that means an object is thrown when we throw an exception. Moreover only those objects could be thrown whose classes are derived from **Throwable**. It is interesting to note here that the objects of our own design could also be thrown provided that they should be the subclass of some member of the Throwable family. Also the throwable classes which are defined by you must extend **Exception** class.

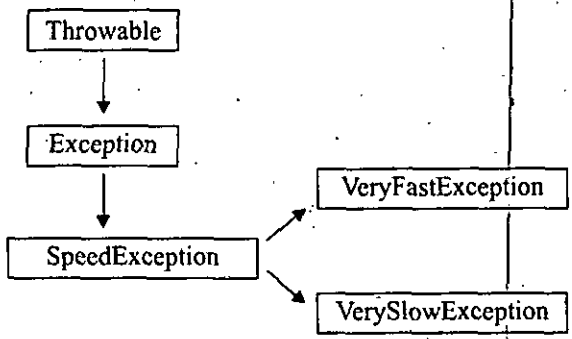
It depends upon the situation that whether to use an existing exception class from java.lang or create any of your own. Such as **IllegalArgumentException,** a subclass of **RuntimeException** in java.lang can be thrown if any method with an invalid argument is thrown by you. On the other hand you need not to worry if you wish to impart some more information about any unusual condition other than a class from java.lang because it will be indicated by the class of exception object itself.

For example, if a thrown exception object has class IllegalArgumentException, that indicates someone passed an illegal argument to a method. Sometimes you will want to indicate that a method encountered an abnormal condition that isn't represented by a class in the Throwable family of java.lang.

For instance, lets tweak an example below that demonstrates the exceptional conditions that might occur while driving a car.

```
// In Source Packet in file except/ex1/SpeedException.java
class SpeedException extends Exception
{ }
// In Source Packet in file except/ex1/VeryFastException.java
class VeryFastException extends SpeedException
{ }
// In Source Packet in file except/ex1/VerySlowException.java
class VerySlowException extends SpeedException {
}
```

Lets tweak the diagram below.



NOTES

It is clear from the above program that there is something abnormal with the speed of the car *i.e.*, either it is very fast or it is very slow. Hence two exceptions are thrown by the program-**VeryFastException** and **VerySlowException.** To be more precise the **SpeedException** family specifies three new exceptions thrown by the program which indicate some abnormal conditions. That is the SpeedException specifies that there is something unusual with the speed; VeryFastException and VerySlowException specifies the abnormal conditions of the speed.

4.2. USING TRY AND CATCH

Exception, that means exceptional errors. Actually *exceptions* are used for handling errors in programs that occurs during the program execution. 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.

In the following example code you will see that how the exception handling can be done in java program. This example reads two integer numbers for the variables a and b. If you enter any other character except number (0 – 9) then the error is caught by *NumberFormatException* object. After that ex.getMessage () prints the information about the error occurring causes.

Code of the program:

```
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());
```

NOTES

```
    }
    catch(NumberFormatException ex){
      System.out.println(ex.getMessage( )
                                  + " is not a numeric value.");
      System:exit(0);
    } }}
```

Catching Exceptions: A method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code and the syntax for using try/catch looks like the following:

```
try
{
  //Protected code
}catch(ExceptionName e1)
{ //Catch block
}
```

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follow the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

Example: *The following is an array is declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.*

```
// File Name : ExcepTest.java
import java.io.*;
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");
    }}
```

This would produce following result:

Exception thrown: java.lang.ArrayIndexOutOfBoundsException: 3 Out of the block

To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a **try** block. Immediately following the **try** block, include a **catch** clause that specifies the exception type that you wish to catch. To illustrate how easily this can be done, the following program includes a **try** block and a **catch** clause which processes the **ArithmeticException** generated by the division-by-zero error:

NOTES

```
class Exc2 {
public static void main(String args[]) {
int d, a;
try { // monitor a block of code.
d = 0;
a = 42 / d;
System.out.println("This will not be printed.");
} catch (ArithmeticException e) { // catch divide-by-zero error
System.out.println("Division by zero.");
}
System.out.println("After catch statement.");
}}
```

This program generates the following output:

Division by zero.

After catch statement.

Notice that the call to **println( )** inside the **try** block is never executed. Once an exception is thrown, program control transfers out of the **try** block into the **catch** block. Put differently, **catch** is not "called," so execution never "returns" to the **try** block from a **catch**. Thus, the line "This will not be printed." is not displayed. Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try/catch** mechanism.

A **try** and its **catch** statement form a unit. The scope of the **catch** clause is restricted to those statements specified by the immediately preceding **try** statement. A **catch** statement cannot catch an exception thrown by another **try** statement. The statements that are protected by **try** must be surrounded by curly braces. You cannot use **try** on a single statement.

The goal of most well-constructed **catch** clauses should be to resolve the exceptional condition and then continue on as if the error had never happened. For example, in the next program each iteration of the **for** loop obtains two random integers. Those two integers are divided by each other, and the result is used to divide the value 12345. The final result is put into **a**. If either division operation causes a divide-by-zero error, it is caught, the value of **a** is set to zero, and the program continues.

```
// Handle an exception and move on.
import java.util.Random;
class HandleError {
public static void main(String args[]) {
int a=0, b=0, c=0;
Random r = new Random();
for(int i=0; i<32000; i++) {
try { b = r.nextInt();
c = r.nextInt();
a = 12345 / (b/c);
} catch (ArithmeticException e) {
System.out.println("Division by zero.");
a = 0;   // set a to zero and continue
} System.out.println("a: " + a);
}}}
```

NOTES

4.3. HANDLING MULTIPLE EXCEPTIONS

A single try block can have many catch blocks. This is necessary when the try block has statements that raise different types of exceptions. For *e.g.*, the following code traps three types of exceptions:

```
Public class TryCatch
{ public static void main(String args[ ])
{ int array[] = {0,0};
  int num1,num2,result = 0;
  num1=100 ; num2=0;
  try
  { result = num1 / num2;
  System.out.println(num1 / array[2]);
  }catch (ArithmeticException e)
  { System.out.println ("Error….Division by zero");
  } catch (ArrayIndexOutOfBoundsException e)
  { System.out.println ("Error…. of Bounds");
  }catch (Exception e)
  { System.out.println("Error…");
  } System.out.println ("The result is : "+result);
  }}
```

4.4. SEQUENCING CATCH BLOCKS

If an exception is thrown during a sequence of statements inside a try-catch block, the sequence of statements is interrupted and the flow of control will skip directly to the catch-block. This code can be interrupted by exceptions in several places:

```
public void openFile()
{ try {
// constructor may throw FileNotFoundException
FileReader reader = new FileReader("someFile");
int i =0;
while(i!= -1)
{ //reader.read() may throw IOException
i = reader.read();
System.out.println((char) i );
} reader.close();
System.out.println("– File End –");
} catch (FileNotFoundException e) {
//do something clever with the exception
} catch (IOException e) {
//do something clever with the exception
}}
```

NOTES

If the **reader.read()** method call throws an IOException, the following System.out.println((char) i); is not executed. Neither is the last **reader.close()** or the System.out.println("— File End —"); statements. Instead the program skips directly to the catch(IOException e){ ... } catch clause. If the new FileReader("someFile"); constructor call throws an exception, none of the code inside the try-block is executed.

4.5. USING FINALLY

When exception is thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, the method may return prematurely. For example, if a method opens a database connection on entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism the finally keyword is designed to address this contingency.

Finally creates a block of code that will be executed after a **try/catch** block has completed and before the code following the **try/catch** block. The **finally** block will execute whether or not an exception is thrown. If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception. Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns. This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. The **finally** clause is optional. However, each **try** statement requires at least one **catch** or a **finally** clause. Here is an example program that shows three methods that exit in various ways, none without executing their **finally** clauses:

```
// Demonstrate finally.
class FinallyDemo {
// Through an exception out of the method.
static void procA() {
try { System.out.println("inside procA");
throw new RuntimeException("demo");
}finally {
System.out.println("procA's finally");
}}
// Return from within a try block.
static void procB() {
try {
System.out.println("inside procB");
return;
} finally {
System.out.println("procB's finally");
}
}
// Execute a try block normally.
static void procC() {
try {
```

```
System.out.println("inside procC");
} finally {
System.out.println("procC's finally");
} }
public static void main(String args[]) {
try {
procA();
} catch (Exception e) {
System.out.println("Exception caught");
}procB();
procC();
}}
```

Output:

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```

In this example, **procA( )** prematurely breaks out of the **try** by throwing an exception. The **finally** clause is executed on the way out. **procB( )**'s **try** statement is exited via a **return** statement. The **finally** clause is executed before **procB( )** returns. In **procC( )**, the **try** statement executes normally, without error. However, the **finally** block is still executed.

Note: If a finally block is associated with a try, the finally block will be executed upon conclusion of the try.

4.6. BUILT-IN EXCEPTION

The built-in exceptions in java are categorized on the basis of whether the exception is handled by the java compiler or not. Java consists of the following categories of built-in exceptions:

- Checked Exceptions.
- Unchecked Exceptions.

***Checked Exceptions*:** Checked exceptions are the objects of the Exception class or any of its subclasses excluding the Runtime Exception class. Checked Exceptions are the invalid conditions that occur in a java program due to invalid user input, network connectivity problem or database problems. Java uses the try-catch block to handle the checked exceptions. The statements within a program that throw an exception are placed in the try block. You associate an exception-handler with the try block by providing one or more catch handlers immediately after the try block. Various checked exceptions defined in the java.lang.package are:

1. ClassNotFoundException.
2. IllegalAccessException.

3. InstantiationException.
4. NoSuchMethodException.

Unchecked Exceptions: Unchecked exceptions are the run-time errors that occur because of programming errors, such as invalid arguments passed to a public method. The java complier does not check the unchecked exceptions during program compilation. Various UncheckedExceptions are:

1. ArithmeticException.
2. ArrayIndexOutOfBoundsException.
3. ArrayStoreException.
4. ClassCastException.
5. IllegalArgumentException.
6. NegativeArraySizeException.
7. NullPointerException.
8. NumberFormatException.

Java defines several exception classes inside the standard package **java.lang**. The most general of these exceptions are subclasses of the standard type RuntimeException. 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. Following is the list of Java Unchecked RuntimeException.

Exception	***Description***
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	*Illegal monitor operation, such as waiting on an* unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
UnsupportedOperationException	An unsupported operation was encountered.

Following is the list of Java Checked Exceptions Defined in java.lang.

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

4.7. THROWING EXCEPTIONS

We can throw an exception explicitly using the throw statement. For *e.g.*, we need to throw an exception when a user enters a wrong login ID or password. The throws clause is used to list the types of exception that can be thrown during the execution of a method in a program.

Using the Throw Statement

1. The throw statement causes termination of the normal flow of control of the java code and stops the execution of the subsequent statements.
2. The throw clause transfers the control to the nearest catch block handling the type of exception object throws.
3. If no such catch block exists, the program terminates.

The throw statement accepts a single argument, which is an object of the Exception class Syntax to declare the throw statement,

```
throw ThrowableObj
```

We can use the following code to throw the IllegalStateException exception:

```
class ThrowStatement
{ Static void throwdemo ( )
{ try { throw new IllegalStateException ( );
} catch (NullPointerException objA)
{ System.out.println ("Not Caught by the catch block inside
                                                    throwDemo ().");
} } public static void main (String args[ ])
{ try { throwDemo ( );
} Catch (IllegalStateException objB)
{ System.out.println ("Exception Caught in:"+objB);
}} }
```

NOTES

Output:

```
javac ThrowStatement.java
java ThrowStatement
Exception Caught in: java.lang.IllegalStateException
```

The programmer can throw an Exception from a method using the statement throw. A call to the above mentioned sample method should be always placed in a try block as it is throwing a Checked Exception–IOException.

Example:

```
public void sample(){
//Statements
//if (somethingWrong)
IOException e = new IOException();
throw e;
//More Statements
}
```

In Java, to invoke a method, the programmer needs to know the name of the method, list of parameters, the return type and the Checked Exceptions being thrown by the method. Java syntax mandates the programmer to list all the Checked Exceptions that are being thrown from a method using the throws clause. If method is trying to throw an exception, then the programmer must handle the exception either with try/catch block or with throws clause. In case a method throws more than one exception, all of them should be listed in throws.

Example:

```
class Number{
private int num;
public void accept(int n) throws Exception{
if( n == 0 ){
throw new Exception("can't assign zero…");
}
else if(n < 0){
throw new Exception("can't assign -ve value…");
}
else{
System.out.println("\n\n\tValid value.");
num = n;
}}}
public class ThrowsDemo{
public static void main(String args[]){
Number ob = new Number();
try{
```

```
ob.accept(-8);
//ob.accept(10);
}
catch(Exception e){
System.out.println("\n Error: " + e);
}
System.out.println("\n\n");
}}
```

Result:

```
Error: java.lang.Exception: can't assign -ve value.
```

4.8. CATCHING EXCEPTION

To catch an exception in Java, you write a try block with one or more catch clauses. Each catch clause specifies one exception type that it is prepared to handle. The try block places a fence around a bit of code that is under the watchful eye of the associated catchers. If the bit of code delimited by the try block throws an exception, the associated catch clauses will be examined by the Java virtual machine. If the virtual machine finds a catch clause that is prepared to handle the thrown exception, the program continues execution starting with the first statement of that catch clause.

As an example, consider a program that requires one argument on the command line, a string that can be parsed into an integer. When you have a String and want an int, you can invoke the parseInt() method of the Integer class. If the string you pass represents an integer, parseInt() will return the value. If the string doesn't represent an integer, parseInt() throws NumberFormatException. Here is how you might parse an int from a command-line argument:

```
// In Source Packet in file except/ex1/Example1.java
class Example1 {
  public static void main(String[] args) {
  int temperature = 0;
  if (args.length > 0) {
  try {
  temperature = Integer.parseInt(args[0]);
    }
  catch(NumberFormatException e) {
  System.out.println( "Must enter integer as first argument.");
  return;   } }
    else {
  System.out.println( "Must enter temperature as first
                                                    argument.");
  return;
  }
```

NOTES

```
    // Create a new coffee cup and set the temperature of its
       coffee.
    CoffeeCup cup = new CoffeeCup();
    cup.setTemperature(temperature);
    // Create and serve a virtual customer.
    VirtualPerson cust = new VirtualPerson();
    VirtualCafe.serveCustomer(cust, cup);
    }}
```

Here, the invocation of parseInt() sits inside a try block. Attached to the try block is a catch clause that catches NumberFormatException:

```
catch(NumberFormatException e)
{ System.out.println( "Must enter integer as first argument.");
  return;
}
```

The lowercase character e is a reference to the thrown (and caught) NumberFormatException object. This reference could have been used inside the catch clause, although in this case it isn't. (Examples of catch clauses that use the reference are shown later in this article.)

If the user types Harumph as the first argument to the Example1 program, parseInt() will throw a NumberFormatException exception and the catch clause will catch it. The program will print:

Must enter integer as first argument

Although the above example had only one catch clause, you can have many catch clauses associated with a single try block. Here's an example:

```
// In Source Packet in file except/ex1/VirtualCafe.java
class VirtualCafe {
  public static void serveCustomer(VirtualPerson cust,
  CoffeeCup cup) {
   try {
  cust.drinkCoffee(cup);
  System.out.println("Coffee is just right.");
  }
 catch (TooColdException e) {
 System.out.println("Coffee is too cold.");
 // Deal with an irate customer...
 }
catch (TooHotException e) {
System.out.println("Coffee is too hot.");
 // Deal with an irate customer...
 }}}
```

If any code inside a try block throws an exception, its catch clauses are examined in their order of appearance in the source file. For example, if the try block in the above example throws an exception, the catch clause for TooColdException will be examined first, then the catch clause for TooHotException. During this examination process, the first catch clause encountered that handles the thrown object's class gets to "catch" the exception. The ordering of catch-clause examination matters because it is possible that multiple catch clauses of a try block could handle the same exception.

4.9. USER DEFINED EXCEPTION

Java supports exception handling by its try catch constructs. Exceptions are conditions which the JVM or applications developed in Java are unable to handle. The Java API defines exception classes for such conditions. These classes are derived from java.lang. Throwable class. User defined exceptions are those exceptions which an application provider or an API provider defines by extending java.lang. Throwable class. Though Java provides an extensive set of in-built exceptions, there are cases in which we may need to define our own exceptions in order to handle the various application specific errors that we might encounter.

While defining an user defined exception, we need to take care of the following aspects:

- The user defined exception class should extend from Exception class.
- The toString() method should be overridden in the user defined exception class in order to display meaningful information about the exception.

Let us see a simple example to learn how to define and make use of user defined exceptions.

NegativeAgeException.java

```
public class NegativeAgeException extends Exception {
 private int age;
 public NegativeAgeException(int age){
  this.age = age;
  }
  public String toString(){
  return "Age cannot be negative" + " " +age ;
  }}
```

CustomExceptionTest.java

```
public class CustomExceptionTest {
 public static void main(String[] args) throws Exception{
  int age = getAge();
  if (age < 0){
  throw new NegativeAgeException(age);
  }else{
  System.out.println("Age entered is " + age);
  }  }
 static int getAge(){
  return -10;
 }}
```

NOTES

In the CustomExceptionTest class, the age is expected to be a positive number. It would throw the user defined exception NegativeAgeException if the age is assigned a negative number. At runtime, we get the following exception since the age is a negative number.

Exception in thread "main" Age cannot be negative -10
at tips.basics.exception.CustomExceptionTest.main
(CustomExceptionTest.java:10)

SUMMARY

- A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer.
- A runtime exception is an exception that occurs that probably could have been avoided by the programmer.
- These are not exceptions at all, but problems that arise beyond the control of the user or the programmer.
- Finally creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block. The finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception.
- The built-in exceptions in java are categorized on the basis of whether the exception is handled by the java compiler or not. Java consists of the following categories of built-in exceptions:
- Checked exceptions are the objects of the Exception class or any of its subclasses excluding the Runtime Exception class. Checked Exceptions are the invalid conditions that occur in a java program due to invalid user input, network connectivity problem or database problems.
- Unchecked exceptions are the run-time errors that occur because of programming errors, such as invalid arguments passed to a public method.

REVIEW QUESTIONS

1. What are exception classes? Explain with an hierarchy structure.
2. Explain with an example using try and catch exceptions.
3. How to handle multiple exceptions?
4. Write short notes on the following:
   (*i*) Sequencing catch block.
   (*ii*) using finally.
   (*iii*) Built in exception.
   (*iv*) Throwing exceptions.
   (*v*) Catching exceptions.
5. Differentiate between checked exception and unchecked exceptions.
6. What are user defined exception? Explain with an example.

NOTES

UNIT

5

PACKAGES AND INTERFACES

STRUCTURE

5.0 Learning Objectives
5.1 Creating Packages
5.2 Adding Classes to Existing Packages
5.3 Interface
5.4 Creating Interfaces
5.5 Exceptions
- *Summary*
- *Review Questions*

5.0. LEARNING OBJECTIVES

After going through this unit, you will be able to :

- explain creating packages
- discuss interface
- define exceptions
- explain interface
- discuss adding classes to exsisting packages.

5.1. CREATING PACKAGES

Java is an object oriented programming language and any Java programmer will quickly build a large number of different classes for use in each application that they develop. Initially it may be tempting to store the classes in the same directory as the end application, but as the complexity applications grow so will the number of classes. The answer is to organize the classes into *packages*.

When creating a package, we should choose a name for the package and put a **package** statement with that name at the top of every source file that contains the classes, interfaces, enumerations and annotation types that we want to include in the package. The **package** statement should be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

If a package statement is not used then the class, interfaces, enumerat annotation types will be put into an unnamed package. Let us look at cre a package called **animals**. It is common p use pa s to avoid any conflicts with the names of interfaces

NOTES

```
package animals;
interface Animal
{ public void eat();
  public void travel();
}
```

Now put an implementation in the same package *animals*:

```
package animals;
public class MammalInt implements Animal
{ public void eat(){
  System.out.println("Mammal eats");
  }
  public void travel(){
    System.out.println("Mammal travels");
  }
  public int noOfLegs(){
    return 0;
  }
  public static void main(String args[]){
    MammalInt m = new MammalInt();
    m.eat();
    m.travel();
  }}
```

Now you compile these two files and put them in sub-directory called animals and try to run as follows:

```
package mymath;
class mathops
{public static void add(int i,int j)
{ int k=i+j;
System.out.println("The sum of "+i+" and"+j+" is "+k);
}
public void sub(int i,int j)
{ int k=i-j;
System.out.println("The substaction of "+i+" and"+j+" is "+k);
```

NOTES

```
}
public static void mul(int i,int j)
{ int k=i*j;
System.out.println("The multiplication of "+i+" and"+j+" is "+k);
}
public  void div(int i,int j)
{ int k=i/j;
System.out.println("The division of "+i+" and"+j+" is "+k);
}}
```

How to create a package: Suppose we have a file called **HelloWorld.java,** and we want to put this file in a package world. First thing we have to do is to specify the keyword package with the name of the package we want to use (world in our case) on top of our source file, before the code that defines the real classes in the package, as shown in our HelloWorld class below:

```
// only comment can be here
package world;
public class HelloWorld
{ public static void main(String[] args)
{   System.out.println("Hello World");
 }}
```

One thing is important that after creating a package for the class is to create nested subdirectories to represent package hierarchy of the class. In our case, we have the **world** package, which requires only one directory. So, we create a directory **world** and put our HelloWorld.java into it.

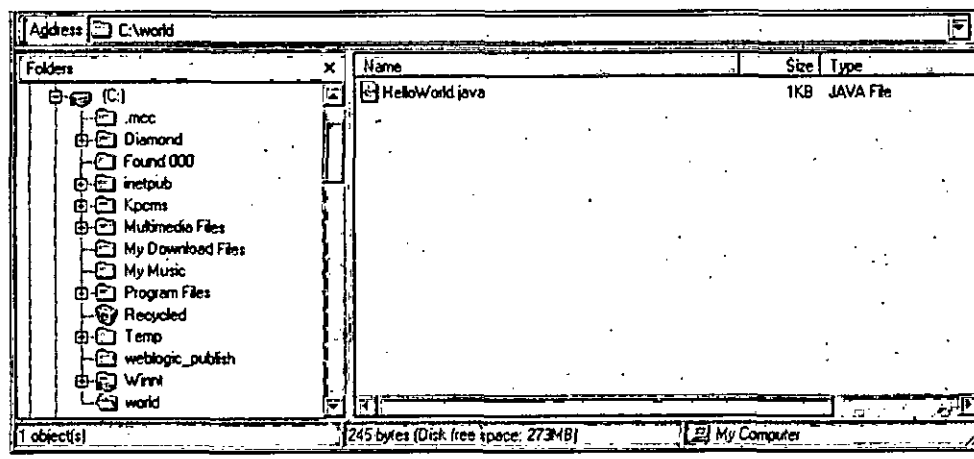


Fig. 5.1. HelloWorld in world package (C:\world\HelloWorld.java)

That's it!!! Right now we have HelloWorld class inside world package. Next, we have to introduce the world package into our **CLASSPATH**.

Setting up the CLASSPATH: From Fig. 5.1 we put the package world under C:. So we just set our CLASSPATH as:

NOTES

```
set CLASSPATH=.;C:\;
```

We set the CLASSPATH to point to 2 places, . (dot) and C:\ directory.

When compiling HelloWorld class, we just go to the world directory and type the command:

```
C:\world\javac HelloWorld.java
```

If you try to run this HelloWorld using **java HelloWorld**, you will get the following error:

```
C:\world>java HelloWorld
Exception in thread "main" java.lang.NoClassDefFoundError:
HelloWorld (wrong name:
world/HelloWorld).
```

To make this example more understandable, let's put the HelloWorld class along with its package (world) be under **C:\myclasses** directory instead. The new location of our HelloWorld should be as shown in Fig. 5.2.

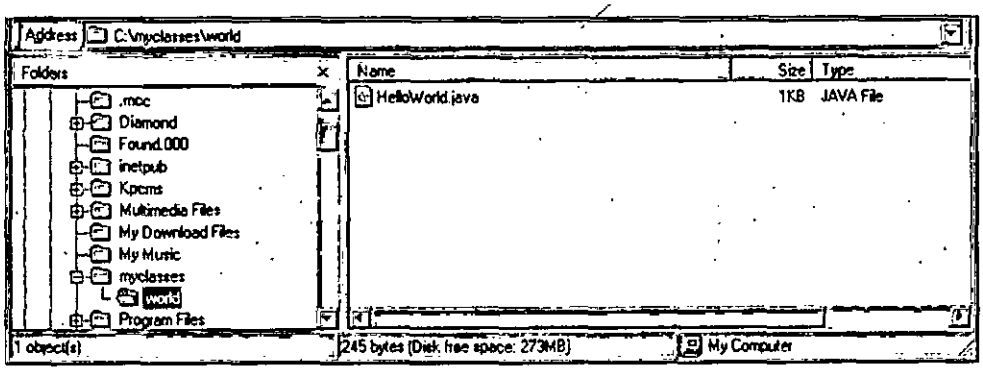


Fig. 5.2. HelloWorld class (in world package) under myclasses directory

We just changed the location of the package from C:\world\HelloWorld.java to C:\myclasses\world\HelloWorld.java. Our CLASSPATH then needs to be changed to point to the new location of the package world accordingly.

```
set CLASSPATH=.;C:\myclasses;
```

Thus, Java will look for java classes from the current directory and **C:\myclasses** directory instead. We can run the HelloWorld from anywhere as long as we still include the package world in the CLASSPATH. For example,

```
C:\>set CLASSPATH=.;C:\;
C:\>set CLASSPATH  // see what we have in CLASSPATH
CLASSPATH=.;C:\;
C:\>cd world
C:\world>java world.HelloWorld
Hello World
C:\world>cd ..
C:\>java world.HelloWorld
Hello World
```

NOTES

Subpackage (package inside another package)

Assume we have another file called **HelloMoon.java**. We want to store it in a subpackage **"moon"**, which stays inside package **world**. The HelloMoon class should look something like this:

```
package world.moon;
public class HelloMoon {
  private String holeName = "rabbit hole";
  public getHoleName() {
    return hole;
  }
   public setHole(String holeName) {
    this.holeName = holeName;
  } }
```

5.2. ADDING CLASSES TO EXISTING PACKAGES

- In order to add a class to an existing package just adds the package statement as the first statement before the class definition.
- Save the file in the folder that contains other classes of that package.
- Compile the file to generate the class file.
- Now the new class is added into the package and can be imported into the other programs.

To add a Class to an existing Java Package do the following:

To add the class **MyClass** to the Java Package **v2k.awt**, add the following statement to the MyClass.java source file. (It must be the first non-comment statement of the file):

MyClass.java:

```
package v2k.awt;
```

Move the Java source file MyClass.java to the source directory for Package **v2k.awt**:

UNIX example:

```
% mv MyClass.java /GUI/java/src/v2k/awt/.
```

You are ready to compile the new class and make it part of the **v2k.awt** Java Package. If the CLASSPATH environment variable *is* set (the path to search for classes being imported by MyClass.java), do the following:

% javac -d /GUI/java/classes /GUI/java/src/v2k/awt/MyClass.java

The *-d* option tells the compiler to place the MyClass.class file in the directory: **/GUI/java/classes/v2k/awt**.

If the CLASSPATH environment is *not* set, do the following:

% javac -d /GUI/java/classes -classpath .:/GUI/java/classes:/usr/java/classes /GUI/java/src/v2k/awt/MyClass.java

The *-d* option tells the compiler to place the MyClass.class file in the directory: **/GUI/java/classes/v2k/awt**.

The ***-classpath*** option tells the compiler to search for classes being imported by MyClass.java in the path: **.:/GUI/java/classes/**.

The class "MyClass" is now part of the Java Package v2k.awt. We can now import this class using the statement *import v2k.awt.MyClass*.

NOTES

5.3. INTERFACE

An **interface** in the Java programming language is an abstract type that is used to specify an interface that classes must implement. Interfaces are declared using the **interface** keyword and may only contain method signatures and constant declarations. An interface may never contain method definitions.

Interfaces cannot be instantiated. A class that implements an interface must implement all of the methods described in the interface, or be an abstract class. Object references in Java may be specified to be of an interface type; in which case, they must either be null, or be bound to an object that implements the interface.

One benefit of using interfaces is that they simulate multiple inheritance. All classes in Java (other than java.lang.Object, the root class of the Java type system) must have exactly one base class; multiple inheritance of classes is not allowed. Furthermore, a Java class may implement and an interface may extend, any number of interfaces; however an interface may not implement an interface.

Interfaces are used to encode similarities which the classes of various types share, but do not necessarily constitute a class relationship. For instance, a human and a parrot can both whistle; however, it would not make sense to represent Humans and Parrots as subclasses of a Whistler class. Rather they would most likely be subclasses of an Animal class, but both would implement the Whistler interface.

Another use of interfaces is being able to use an object without knowing its type of class, but rather only that it implements a certain interface. For instance, if one were annoyed by a whistling noise, one may not know whether it is a human or a parrot, because all that could be determined is that a whistler is whistling. In a more practical example, a sorting algorithm may expect an object of type Comparable. Thus, it knows that the object's type can somehow be sorted, but it is irrelevant what the type of the object is whistle() will call the implemented method whistle of object whistler no matter what class it has, provided it implements Whistler.

For example:

```
interface Bounceable
{ void setBounce();  // Note the semicolon:
                     // Interface methods are public and abstract.
                     // Think of them as prototypes only; no
                        implementations are allowed.
}
```

NOTES

Defining an interface: Interfaces are defined with the following syntax (compare to Java's class definition):

```
[visibility] interface InterfaceName [extends other interfaces]
{
   constant declarations
   abstract method declarations
}
```

The body of the interface contains abstract methods, but since all methods in an interface are, by definition, abstract, the abstract keyword is not required. Since the interface specifies a set of exposed behaviors, all methods are implicitly public.

Thus, a simple interface may be

```
public interface Predator {
    boolean chasePrey(Prey p);
    void eatPrey(Prey p);
}
```

The member type declarations in an interface are implicitly static, final and public, but otherwise they can be any type of class or interface. The syntax for implementing an interface uses this formula:

```
... implements InterfaceName[, another interface, another, ...]
                                                              ...

Classes may implement an interface. For example,
public class Lion implements Predator {

   public boolean chasePrey(Prey p)
 {     // programming to chase prey p (specifically for a lion)
    }

    public void eatPrey (Prey p)
{       // programming to eat prey p (specifically for a lion)
     }}
```

If a class implements an interface and does not implement all its methods, it must be marked as abstract. If a class is abstract, one of its subclasses is expected to implement its unimplemented methods.

Classes can implement multiple interfaces

```
public class Frog implements Predator, Prey { ... }
```

Interfaces are commonly used in the Java language for callbacks. Java does not allow the passing of methods (procedures) as arguments. Therefore, the practice is to define an interface and use it as the argument and use the method signature knowing that the signature will be later implemented.

Subinterfaces: Interfaces can extend several other interfaces, using the same formula as described below. For example

```
public interface VenomousPredator extends Predator, Venomous
{       //interface body
}
```

is legal and defines a subinterface. Note how it allows multiple inheritances, unlike classes. Note also that predator and venomous may possibly define or inherit methods with the same signature. When a class implements VenomousPredator it will implement both methods simultaneously.

Examples: Some common Java interfaces are:

- Comparable has the method compareTo, which is used to describe two objects as equal, or to indicate one is greater than the other. Generics allow implementing classes to specify which class instances can be compared to them.
- Serializable is a marker interface with no methods or fields - it has an empty body. It is used to indicate that a class can be serialized. Its Javadoc describes how it should function, although nothing is programmatically enforced.

```
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");
  }
}
```

5.4. CREATING INTERFACES

Interface is defined as group of method, that implement a empty body. An example of Radio Tuner, when a listener switch on Radio Tuner, the tuner act as interface between the electrical wing circuit inside the radio and you. Usually the Java do not support multiple inheritance, Interface in java is used for multiple inheritance.

Understand with Example: The given below code illustrates that interface in java support multiple inheritance in order to implement interface we implement a keyword implement. The step involved in the program are described below:

NOTES

NOTES

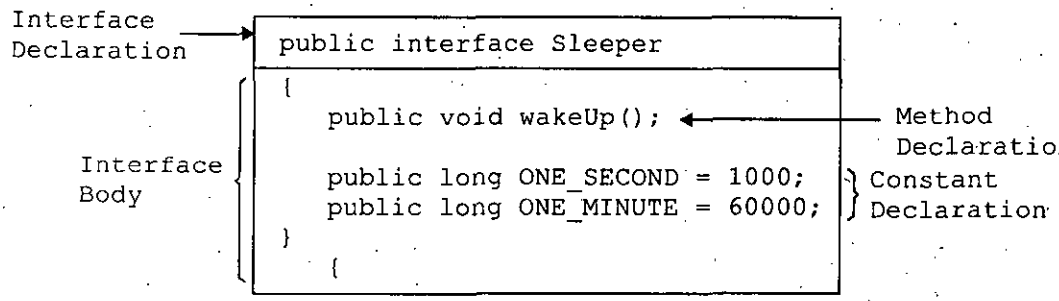
1. We have taken an interface check, that implement a empty method message ( ).
2. Inside the class we create an object of interface check and implement a body to the method message.
3. The object call and expose a message ( ) method to the outside world.
4. In case there is an exception in the try block, the subsequent catch block handle the exception.

```
interface check {
public void message();
}
public class Interface {
public static void main(String[] args) {
  try {
  check t = new check() {
  public void message() {
  System.out.println("Method defined in the interface");
  }};
  t.message();
  }
  catch (Exception ex)
  {  System.out.println("" + ex.getMessage());
  } } }
```

Output of the program:

Method defined in the interface

The following figure shows that an interface definition has two components: the interface declaration and the interface body. The interface declaration declares various attributes about the interface such as its name and whether it extends another interface. The interface body contains the constant and method declarations within that interface.



The Interface Declaration: The declaration for the Sleeper interface uses the two required elements of an interface declaration—the interface keyword and the name of the interface—plus the public access modifier:

```
public interface Sleeper {
   . . .
}
```

NOTES

An interface declaration can have one other component: the public access specifier and a list of super interfaces. The full interface declaration looks like this figure:

```
public                          Makes this interface public.
interface InterfaceName         Class cannot be instantiated.
Extends SuperInterfaces         This interface's superinterfaces:

{
      InterfaceBody
}
```

The public access specifier indicates that the interface can be used by any class in any package. If you do not specify that your interface is public, then your interface will be accessible only to classes that are defined in the same package as the interface.

To create an interface definition:

- define it like a Java class, in its own file that matches the interface name
- use the keyword interface instead of class.
- declare methods using the same approach as abstract methods:
  - note the semicolon after each method declaration-and that no executable code is supplied(and no curly braces).
  - the elements will automatically be public and abstract, and cannot have any other state; it is OK to specify those terms, but not necessary (usually public is specified and abstract is not-that makes it easy to copy the list of methods, paste them into a class, and modify them).
- The access level for the entire interface is usually public:
  - it may be omitted, in which case the interface is only available to other classes in the same package (*i.e.*, in the same directory).
  - note, for the sake of completeness, there are situations where the interface definition could be protected or private; these involve what are called *inner classes.*

Syntax:

```
[modifiers] interface InterfaceName {
 // declaring methods
[public abstract] returnType methodName1(arguments);
 // defining constants
[ public static final ]
 type propertyName = value;
}
```

5.5. EXCEPTIONS

In java it is possible to define two categories of Exceptions and Errors.

- **JVM Exceptions:** These are exceptions/errors that are exclusively or logically thrown by the JVM. Examples: NullPointerException, ArrayIndexOutOf BoundsException, ClassCastException.
- **Programmatic exceptions**: These exceptions are thrown explicitly by the application or the API programmers Examples: IllegalArgumentException, IllegalStateException.

Provides classes and interfaces for handling exceptions thrown by the SDK layer.

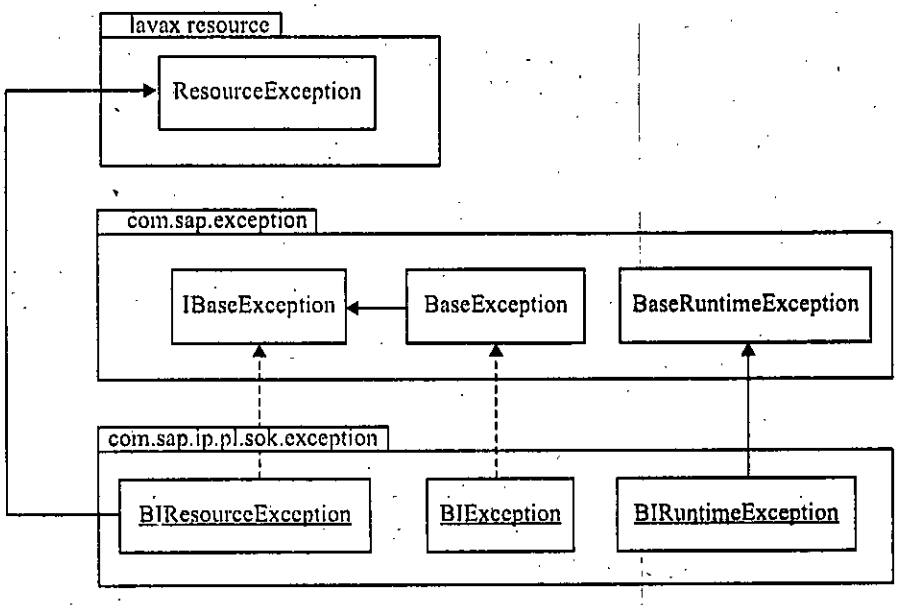
Exception Summary	
BICapabilityNotSupportedException	Thrown when a certain feature is not supported by a connector.
BIConnectionFailedException	Thrown to indicate that a connection has not been successfully established.
BIConnectorException	Thrown to indicate miscellaneous error situations in a connector.
BIException	General-purpose exception for the BI Java SDK.
BIJmiVerifyException	Exception thrown by generated JMI objects of the BI Java SDK.
BIOlapQueryException	Checked exception which is thrown within the context of an OLAP query.
BIOlapQueryRuntimeException	Runtime exception which is thrown within the context of an OLAP query.
BIResourceException	Thrown to indicate a lower-level problem.
BIRuntimeException	Represents BI Java SDK-specific runtime exceptions.
BISQLException	Exception providing information on errors occurring while interacting with BI Java SDK relational connectors.
BISQLWarning	Exception providing information on errors occurring while interacting with BI Java SDK relational connectors.

Provides classes and interfaces for handling exceptions thrown by the SDK layer. Exception handling in the BI Java SDK both conforms to SAP standards and supports legacy exceptions. All exceptions thrown by the SDK interfaces implement the IBaseException interface in order to comply with SAP solution production standards. This also provides integration with SAP's Java logging and tracing framework used for instrumenting the code in case of error situations.

The SDK exceptions either directly implement the IBaseException interface, as in the case of BIResourceException, or they extend exceptions in the package com.sap.exception, as with BaseException. Legacy exceptions, which are exceptions that are thrown by components outside the SDK, do not implement the IBaseException interface but are handled by providing wrapper exceptions. These wrappers have exactly the same semantics as the underlying legacy exception and simply add the implementation of IBaseException. For example, the legacy exception BIResourceException wraps exceptions of the type **javax.resource.ResourceException.**

The diagram below illustrates how the BI Java SDK exceptions relate to legacy exceptions such as javax.resource.RescourceException and the SAP Exception Framework base exceptions:

NOTES



SUMMARY

- Java is an object oriented programming language and any Java programmer will quickly build a large number of different classes for use in each application that they develop.
- When creating a package, we should choose a name for the package and put a package statement with that name at the top of every source file that contains the classes, interfaces, enumerations and annotation types that we want to include in the package.
- In order to add a class to an existing package just adds the package statement as the first statement before the class definition.
- An interface in the Java programming language is an abstract type that is used to specify an interface that classes must implement. Interfaces are declared using the interface keyword and may only contain method signatures and constant declarations. An interface may never contain method definitions.
- One benefit of using interfaces is that they simulate multiple inheritance.
- Interfaces are used to encode similarities which the classes of various types share, but do not necessarily constitute a class relationship.
- JVM Exceptions are exceptions/errors that are exclusively or logically thrown by the JVM.
- Programmatic exceptions are thrown explicitly by the application or the API programmers.

NOTES

REVIEW QUESTIONS

1. How you can creating a packages in Java? Explain with an example.
2. How the class can be added to an existing packages?
3. What is interface? Explain with an example.
4. How you can define an interface?
5. What is subinterfaces? What are the common java interfaces?
6. How you can create an interface? Explain with an example.
7. What are exceptions?

UNIT

6

MULTITHREADED PROGRAMMING

STRUCTURE

6.0 Learning Objectives
6.1 Multithreading an Introduction
6.2 The main Thread
6.3 Java Thread Model
6.4 Thread Priorities
6.5 Synchronization in Java
6.6 Interthread Communication
- *Summary*
- *Review Questions*

6.0. LEARNING OBJECTIVES

After going through this unit, you will be able to :

- explain multithreading
- discuss thread priorities
- derive synchronization in java
- explain interthread communication
- discuss java thread model.

6.1. MULTITHREADING AN INTRODUCTION

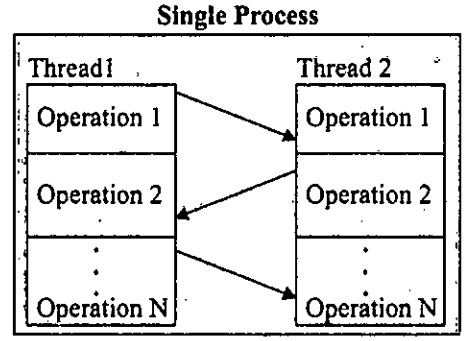
We know about a single thread. Let us know about the concept of multithreading and learn the implementation of it. But before that, let's be aware from the multitasking.

Multitasking: Multitasking allow to execute 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. Multitasking allow processes (*i.e.*, programs) to run concurrently on the program. For Example running the spreadsheet program and you are working with word processor also. Multitasking is running heavyweight processes by a single OS.

NOTES

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. For 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. It allows a program to be more responsible to the user. When a program contains multiple threads then the CPU can switch between the two threads to execute them at the same time. For example, look at the diagram shown as:

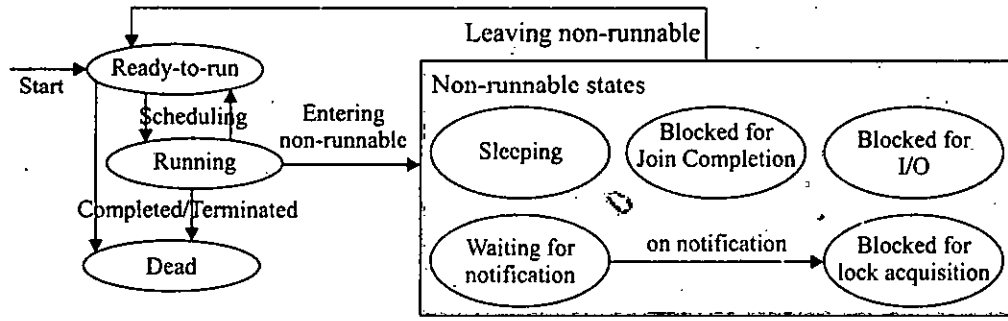


In this diagram, two threads are being executed having more than one task. The task of each thread is switched to the task of another thread.

Advantages of multithreading over multitasking:

- Reduces the computation time.
- Improves performance of an application.
- Threads share the same address space so it saves the memory.
- Context switching between threads is usually less expensive than between processes.
- Cost of communication between threads is relatively low.

Different states implementing Multiple-Threads are:



As we have seen different states that may be occur with the single thread. A running thread can enter to any non-runnable state, depending on the circumstances. A thread cannot enters directly to the running state from non-runnable state, firstly it goes to runnable state. Now let's understand the some non-runnable states which may be occur handling the multithreads.

NOTES

- **Sleeping:** On this state, the thread is still alive but it is not runnable, it might be return to runnable state later, if a particular event occurs. On this state a thread sleeps for a specified amount of time. You can use the method **sleep( )** to stop the running state of a thread.

  static void sleep(long millisecond) throws Interrupted Exception

- **Waiting for Notification:** A thread waits for notification from another thread. The thread sends back to runnable state after sending notification from another thread.

  final void wait(long timeout) throws InterruptedException

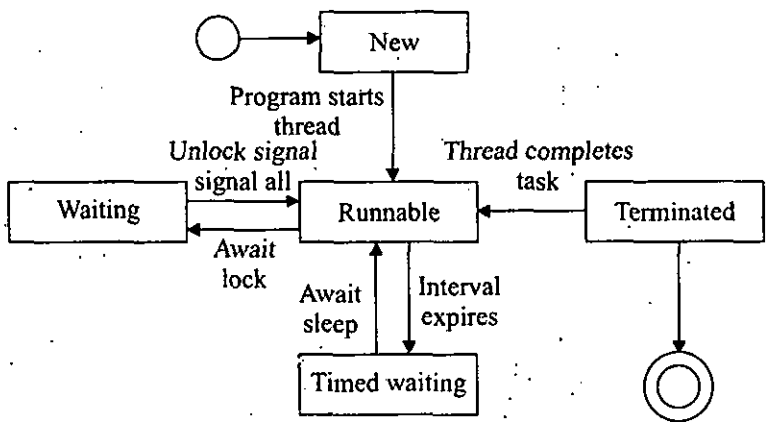
  final void wait(long timeout, int nanos) throws InterruptedException

  final void wait() throws InterruptedException

- **Blocked on I/O:** The thread waits for completion of blocking operation. A thread can enter on this state because of waiting I/O resource. In that case the thread sends back to runnable state after availability of resources.
- **Blocked for joint completion:** The thread can come on this state because of waiting the completion of another thread.
- **Blocked for lock acquisition:** The thread can come on this state because of waiting to acquire the lock of an object.

Short summary of Multithreading: Many applications can run on the same computer at the same time. Each instance, in the running condition, is known as process. Each process can have one or more threads. A thread is a sequence of code, this code is often responsible for one aspect of the program, or one task a program has been given. For instance a program doing a complex long calculation may split into two threads, one to keep a user interface responsive, and one (or more) to progress through the lengthy calculation.

Java has a built in support for multithreading programming. A multithreading is a specialized form of multitasking. Multitasking threads require less overhead than multitasking processes. A thread has various stages during it's life cycle. For example, a thread is born, started, runs, and then dies. Given below diagram contains various stages during it's life cycle:



The description of the stages of a thread during it's life cycle are given below:

Life Cycle Stage	*Description*
New	A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
Runnable	After thread born in 'New' Stage, the thread becomes runnable. A thread in this state is considered to be executing its task.
Waiting	A runnable thread can enter the timed waiting state for a specified interval of time. A thread in these state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
Timed Waiting	A runnable thread can enter the timed waiting state for a specified interval of time. A thread in these state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
Terminated	A runnable thread enters the terminated state when it completes its task or otherwise terminates

6.2. THE MAIN THREAD

The 'main()' method in Java is referred to the thread that is running, whenever a Java program runs. It calls the main thread because it is the first thread that starts running when a program begins. Other threads can be spawned from this main thread. The main thread must be the last thread in the program to end. When the main thread stops, the program stops running. Main thread is created automatically, but it can be controlled by the program by using a Thread object. The Thread object will hold the reference of the main thread with the help of currentThread() method of the Thread class. Listing shows how main thread can be controlled by a program:

```
class MainThread
{
public static void main(String args [])
{
  Thread t = Thread.currentThread ( );
System.out.println ("Current Thread: " + t);
  System.out.println ("Name : " + t.getName ( ));
  System.out.println (" ");
  t.setName ("New Thread");
  System.out.println ("After changing name");
  System.out.println ("Current Thread: " + t);
  System.out.println ("Name: " + t.getName ( ));
  System.out.println (" ");
```

```
System.out.println ("This thread prints first 10 numbers");
try
{ for (int i=1; i<=10;i++)
  { System.out.print(i);
    System.out.print(" ");
    Thread.sleep(1000);
  } }
catch (InterruptedException e)
{ System.out.println(e);
} } }
```

The output of this program is given below:

Current thread: Thread [main , 5,main]

Name : main

After changing name

Current Thread : Thread [New Thread, 5,main]

Name : New Thread

This thread prints first 10 numbers

1 2 3 4 5 6 7 8 9 10

Let us now understand the working of the program in the listing. The program first creates a Thread object called 't' and assigns the reference of current thread (main thread) to it. So now main thread can be accessed via Thread object 't'. This is done with the help of currentThread() method of Thread class which return a reference to the current running thread. The Thread object 't' is then printed as a result of which you see the output Current Thread : Thread [main,5,main]. The first value in the square brackets of this output indicates the name of the thread, the name of the group to which the thread belongs. The program then prints the name of the thread with the help of getName() method.

The name of the thread is changed with the help of setName() method. The thread and thread name is then again printed. Then the thread performs the operation of printing first 10 numbers. When you run the program you will that the system wait for sometime after printing each number. This is caused by the statement Thread.sleep (1000). The sleep() method can also used in another way in which it accepts two arguments. The first argument represents the amount of milliseconds and the second argument represents the amount of nanoseconds. Effectively, the thread will sleep for specified milliseconds + nanoseconds. Its (sleep() method's) prototype is as follows:

Static void sleep (long milliseconds, int nanoseconds)

The Main thread of java can also explain in the following manner:

When a Java program starts up, one thread begins running immediately. This is usually called the *main thread* of your program, because it is the one that is executed when your program begins. The main thread is important for two reasons:

- It is the thread from which other "child" threads will be spawned.
- It must be the last thread to finish execution. When the main thread stops, your program terminates.

NOTES

Although the main thread is created automatically when your program is started, it can be controlled through a **Thread** object. We must obtain a reference to it by calling the method **currentThread( )**, which is a **public static** member of **Thread**. Its general form is shown here: **static Thread currentThread( )**

This method returns a reference to the thread in which it is called. Once we have a reference to the main thread, we can control it just like any other thread. Let's begin by reviewing the following example:

```
// Controlling the main Thread.
class CurrentThreadDemo
{  public static void main(String args[]) {
Thread t = Thread.currentThread();
System.out.println("Current thread: " + t);
// change the name of the thread
t.setName("My Thread");
System.out.println("After name change: " + t);
try {  for(int n = 5; n > 0; n--)
{ System.out.println(n);
  Thread.sleep(1000);
}
}
catch (InterruptedException e)
{ System.out.println("Main thread interrupted");
}
}}
```

In this program, a reference to the current thread (the main thread, in this case) is obtained by calling **currentThread( )**, and this reference is stored in the local variable t. Next, the program displays information about the thread. The program then calls **setName( )** to change the internal name of the thread. Information about the thread is then redisplayed. Next, a loop counts down from five, pausing one second between each line. The pause is accomplished by the **sleep( )** method. The argument to **sleep( )** specifies the delay period in milliseconds. Notice the **try/catch** block around this loop. The **sleep( )** method in **Thread** might throw an **InterruptedException**. This would happen if some other thread wanted to interrupt this sleeping one. This example just prints a message if it gets interrupted. In a real program, you would need to handle this differently. Here is the output generated by this program:

Current thread: Thread [main,5, main]

After name change: Thread [My Thread,5,main]

5

4

3

2

1

Notice the output produced when **t** is used as an argument to **println ( )**. These displays, in order: the name of the thread, its priority, and the name of its group. By default, the name of the main thread is **main**. Its priority is 5, which is the default value, and **main** is also the name of the group of threads to which this thread belongs. A *thread group* is a data structure that controls the state of a collection of threads as a whole. This process is managed by the particular run-time environment and is not discussed in detail here. After the name of the thread is changed, **t** is again output. This time, the new name of the thread is displayed.

6.3. JAVA THREAD MODEL

The benefit of Java's multithreading is that a thread can pause without stopping other parts of your program. A paused thread can restart. A thread will be referred to as dead when the processing of the thread is completed. After a thread reaches the dead state, then it is not possible to restart it.

The thread exists as an object; threads have several well-defined states in addition to the dead states. These state are:

- **Ready State:** When a thread is created, it doesn't begin executing immediately. You must first invoke start ( ) method to start the execution of the thread. In this process the thread scheduler must allocate CPU time to the Thread. A thread may also enter the ready state if it was stopped for a while and is ready to resume its execution.
- **Running State:** Threads are born to run, and a thread is said to be in the running state when it is actually executing. It may leave this state for a number of reasons, which we will discuss in the next section of this unit.
- **Waiting State:** A running thread may perform a number of actions that will cause it to wait. A common example is when the thread performs some type of input or output operations.

As you can see in Fig. 6.1 given below, a thread in the waiting state can go to the ready state and from there to the running state. Every thread after performing its operations has to go to the dead state.

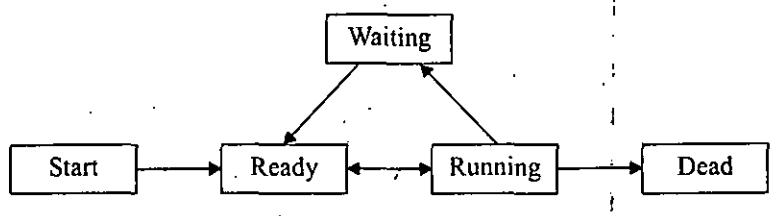


Fig. 6.2. The Thread States

A thread begins as a ready thread and then enters the running state when the thread scheduler schedules it. The thread may be prompted by other threads and returned to the ready state, or it may wait on a resource, or simply stop for sometime. When this happens, the thread enters the waiting state. To run again, the thread must enter the ready state. Finally, the thread will cease its execution and enter the dead state.

The multithreading system in Java is built upon the **Thread Class**, its methods and its companion interface, **Runnable**. To create a new thread, your program will

NOTES

either extend Thread Class or implement the Runnable interface. The Thread Class defines several methods that help in managing threads.

For example, if you have to create your own thread then you have to do one of the following things:

```
1)
class MyThread extends Thread
{
MyThread(arguments) // constructor
{
}//initialization
public void run()
{
// perform operations
}
}
Write the following code to create a thread and start it
running:
MyThread p = new MyThread(arguments);
p.start();

2)
class MyThread implements Runnable
{
MyThread(arguments)
{
//initialization
}
public void run()
{
// perform operation
}
}
```

The Thread Class Constructors

The following are the Thread class constructors:

Thread()

Thread(Runnable target)

Thread (Runnable target, String name)

Thread(String name)

Thread(ThreadGroup group, Runnable target)

NOTES

Thread(ThreadGroup group, Runnable target, String name) Thread(ThreadGroup group, Runnable target, String name, long stackSize) Thread(ThreadGroup group, String name).

Thread Class Method

Some commonly used methods of Thread class are given below:

static Thread **currentThread**() Returns a reference to the currently executing thread object.

String **getName**() Returns the name of the thread in which it is called

int **getPriority**() Returns the Thread's priority

void **interrupt**() Used for Interrupting the thread.

static boolean **interrupted**() Used to tests whether the current thread has been interrupted or not.

boolean **isAlive**() Used for testing whether a tread is alive or not.

boolean **isDaemon**() Used for testing whether a thread is a daemon thread or not.

void **setName**(String NewName) Changes the name of the thread to NewName

void **setPriority**(int newPriority) Changes the priority of thread.

static void **sleep**(long millisec) Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

void **start**() Used to begin execution of a thread .The Java Virtual Machine calls the run method of the thread in which this method is called.

String **toString**() Returns a string representation of thread.String includes the thread's name, priority, and thread group.

static void **yield**() Used to pause temporarily to currently executing thread object and allow other threads to execute.

static int **activeCount**() Returns the number of active threads in the current thread's thread group.

void **destroy**() Destroys the thread without any cleanup.

Creating Threads: Java provides native support for multithreading. This support is centered on the Java.lang.Thread class, the Java.lang.Runnable interface and methods of the Java.lang.object class. In Java, support for multithreaded programming is also provided through synchronized methods and statements.

The thread class provides the capability to create thread objects, each with its own separate flow of control. The thread class encapsulates the data and methods associated with separate threads of execution and enables multithreading to be integrated within Java's object oriented framework. The minimal multithreading support required of the Thread Class is specified by the java.lang.Runnable interface. This interface defines a single but important method run.

```
public void run( )
```

This method provides the entry point for a separate thread of execution. As we have discussed already Java provides two approaches for creating threads. In the first approach, you create a subclass of the Thread class and override the run( ) method to provide an entry point for the Thread's execution. When you create an instance of subclass of Thread class, you invoke start( ) method to cause the thread to execute as

NOTES

an independent sequence of instructions. The start( ) method is inherited from the Thread class. It initializes the thread using the operating system's multithreading capabilities, and invokes the run( ) method.

Now let us see the program given below for creating threads by inheriting the Thread class.

```
//program
class MyThreadDemo extends Thread
{
public String MyMessage [ ]= {"Java","is","very","good",
                                        "Programming","Language"};
MyThreadDemo(String s)
{
super(s);
}
public void run( )
{
String name = getName( );
for ( int i=0; i < MyMessage.length;i++)
{
Wait( );
System.out.println (name +":"+ MyMessage [i]);
}
}
void Wait( )
{
try
{
sleep(1000);
}
catch (InterruptedException e)
{
System.out.println (" Thread is Interrupted");
}
}
}
class ThreadDemo
{
public static void main ( String args [ ])
{
```

```
MyThreadDemo Td1= new MyThreadDemo("thread 1:");
MyThreadDemo Td2= new MyThreadDemo("thread 2:");
Td1.start ( );
Td2.start ( );
boolean isAlive1 = true;
boolean isAlive2 = true;
do
{
if (isAlive1 && ! Td1.isAlive( ))
{
isAlive1= false;
System.out.println ("Thread 1 is dead");
}
if (isAlive2 && !Td2.isAlive( ))
{
isAlive2= false;
System.out.println ("Thread 2 is dead");
}
}
while(isAlive1 || isAlive2);
}
}
```

Output:

thread 1::Java
thread 2::Java
thread 1::is
thread 2::is
thread 1::very
thread 2::very
thread 1::good
thread 2::good
thread 1::Programming
thread 2::Programming
thread 1::Language
thread 2::Language
Thread 1 is dead
Thread 2 is dead

This output shows how two threads execute in sequence, displaying information on the console. The program creates two threads of execution, Td 1 and Td2. The threads display the "Java","is","very","good","Programming","Language" message word

NOTES

by word, while waiting a short amount of time between each word. Because both threads share the console window, the program's output identifies which thread wrote to the console during the program's execution.

Now let us see how threads are created in Java by implementing the java.lang.Runnable interface. The Runnable interface consists of only one method, *i.e.*, run ( )method that provides an entry point into your threads execution.

In order to run an object of class you have created, as an independent thread, you have to pass it as an argument to a constructor of class Thread.

```
//program
class MyThreadDemo implements Runnable
{
public String MyMessage [ ]
      = {"Java","is","very","good","Programming","Language"};
String name;
public MyThreadDemo(String s)
{
name = s;
}
public void run( )
{
for ( int i=0; i < MyMessage.length;i++)
{
Wait( );
System.out.println (name +":"+ MyMessage [i]);
}
}
void Wait( )
{
try
{
Thread.currentThread().sleep(1000);
}
catch (InterruptedException e)
{
System.out.println (" Thread is Interrupted");
}
}
}
class ThreadDemo1
{
```

```
public static void main ( String args [ ])
{
Thread Td1= new Thread( new MyThreadDemo("thread 1:"));
Thread Td2= new Thread(new MyThreadDemo("thread 2:"));
Td1.start ( );
Td2.start ( );
boolean isAlive1 = true;
boolean isAlive2 = true;
do
{
if (isAlive1 && ! Td1.isAlive( ))
{
isAlive1= false;
System.out.println ("Thread 1 is dead");
}
if (isAlive2 && !Td2.isAlive( ))
{
isAlive2= false;
System.out.println ("Thread 2 is dead");
}
}
while(isAlive1 || isAlive2);
}
}
```

Output:

thread 1::Java
thread 2::Java
thread 1::is
thread 2::is
thread 1::very
thread 2::very
thread 1::good
thread 2::good
thread 1::Programming
thread 2::Programming
thread 1::Language
Thread 1 is dead
thread 2::Language
Thread 2 is dead

This program is similar to previous program and also gives same output. The advantage of using the Runnable interface is that your class does not need to extend the thread class. This is a very helpful feature when you create multithreaded applets. The only disadvantage of this approach is that you have to do some more work to create and execute your own threads.

6.4. THREAD PRIORITIES

Java assigns to each thread a priority that determines how that thread should be treated with respect to others. Priorities are integer values. A higher-priority thread *doesn't run any faster than a low-priority thread if it is the only thread running.* Instead, a priority is used to determine when to switch from one running thread to the next. This is called context-switch. A thread can voluntarily relinquish control (yielding, sleeping or blocking on pending I/O). A higher-priority thread (Preemptive multitasking) can preempt another thread.

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**).

Constant	*Description*
Thread.MIN_PRIORITY	The maximum priority of any thread (an int value of 10)
Thread.MAX_PRIORITY	The minimum priority of any thread (an int value of 1)
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 as:

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

When a Java thread is created, it inherits its priority from the thread that created it. At any given time, when multiple threads are ready to be executed, the runtime system chooses the runnable thread with the highest priority for execution. In Java runtime system, **preemptive scheduling** algorithm is applied. If at the execution time a thread with a higher priority and all other threads are runnable then the runtime system chooses the new **higher priority** thread for execution. On the other hand, if two threads of the same priority are waiting to be executed by the CPU then the **round-robin** algorithm is applied in which the scheduler chooses one of them to run according to their round of **time-slice**.

Thread Scheduler: In the implementation of threading scheduler usually applies one of the two following strategies:

- **Preemptive scheduling:** If the new thread has a higher priority then current running thread leaves the runnable state and higher priority thread enter to the runnable state.
- **Time-Sliced (Round-Robin) Scheduling:** A running thread is allowed to be executing for the fixed time, after completion the time, current thread indicates to the thread to enter it in the runnable state.

We can also set a thread's priority at any time after its creation using the **setPriority** method. Let's see, how to set and get the priority of a thread.

```
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);
  int p=cur.getPriority();
  System.out.println("Thread Name
  :"+Thread.currentThread().getName());
  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 of the Program:

```
C:\nisha>javac ThreadPriority.java
C:\nisha>java ThreadPriority
Thread Name     : My Thread 1
Thread Name     : My Thread 2
Thread Priority :Thread[My Thread 2,10,main]
Thread Name     : My Thread 2
Thread Priority :Thread[My Thread 2,10,main]
Thread Name     : My Thread 2
Thread Priority :Thread[My Thread 2,10,main]
Thread Priority :Thread[My Thread 1,
```

In this program two threads are created. We have set up maximum priority for the first thread "**MyThread2**" and minimum priority for the first thread "**MyThread1**" *i.e.,* the after executing the program, the first thread is executed only once and the second thread "**MyThread2**" started to run until either it gets end or another thread of the equal priority gets ready to run state.

6.5. SYNCHRONIZATION IN JAVA

If you want two threads to communicate and share a complicated data structure, such as a linked list or graph, you need some way to ensure that they don't conflict with each other. In other words, you must prevent one thread from writing data while another thread is in the middle of reading it. For this purpose, Java implements model of **interprocess synchronizations.** You know that once a thread enters a monitor, all other threads must wait until that thread exists in the monitor. Synchronization support is built in to the Java language.

There are many situations in which multiple threads must share access to common objects. There are times when you might want to coordinate access to shared resources. For example, in a database system, you would not want one thread to be updating a database record while another thread is trying to read from the database. Java enables you to coordinate the actions of multiple threads using synchronized methods and synchronized statements. Synchronization provides a simple monitor facility that can be used to provide mutual exclusion between Java threads.

Once you have divided your program into separate threads, you need to define how they will communicate with each other. Synchronized methods are used to coordinate access to objects that are shared among multiple threads. These methods are declared with the synchronized keyword. Only one synchronized method at a time

NOTES

can be invoked for an object at a given point of time. When a synchronized method is invoked for a given object, it acquires the monitor for that object. In this case no other synchronized method may be invoked for that object until the monitor is released. This keeps synchronized methods in multiple threads without any conflict with each other.

When a synchronized method is invoked for a given object, it tries to acquire the lock for that object. If it succeeds, no other synchronized method may be invoked for that object until the lock is released. A lock is automatically released when the method completes its execution and returns. A lock may also be released when a synchronized method executes certain methods, such as wait( ).

Now let us see this example which explains how synchronized methods and object locks are used to coordinate access to a common object by multiple threads.

```
//program
class Call_Test
{
synchronized void callme (String msg)
{
//This prevents other threads from entering call( ) while
another thread is using it.
System.out.print ("["+msg);
try
{
Thread.sleep (2000);
}
catch (InterruptedException e)
{
System.out.println ("Thread is Interrupted");
}
System.out.println ("]");
}
}
class Call implements Runnable
{
String msg;
Call_Test ob1;
Thread t;
public Call (Call_Test tar, String s)
{
System.out.println("Inside caller method");
ob1= tar;
msg = s;
```

NOTES

```
t = new Thread(this);
t.start();
}
public void run( )
{
ob1.callme(msg);
}
}
class Synchro_Test
{
public static void main (String args [ ])
{
Call_Test T= new Call_Test( );
Call ob1= new Call (T, "Hi");
Call ob2= new Call (T, "This ");
Call ob3= new Call (T, "is");
Call ob4= new Call (T, "Synchronization");
Call ob5= new Call (T, "Testing");
try
{
ob1.t.join( );
ob2.t.join( );
ob3.t.join( );
ob4.t.join( );
ob5.t.join( );
}
catch ( InterruptedException e)
{
System.out.println (" Interrupted");
}
}
}
```

Output:

Inside caller method
Inside caller method
Inside caller method
Inside caller method
Inside caller method
[Hi]
[This]
[is]
[Synchronization]
[Testing]

If you run this program after removing synchronized keyword, you will find some output similar to:

```
Inside caller method
Inside caller method
Inside caller method
Inside caller method
[Hi[This [isInside caller method
[Synchronization[Testing]
]
]
]
]
```

This result is because when in program sleep( ) is called, the callme( ) method allows execution to switch to another thread. Due to this, output is a mix of the three message strings.

So if at any time you have a method, or group of methods, that manipulates the internal state of an object in a multithreaded situation, you should make these methods synchronized. It helps in avoiding conflict.

An effective means of achieving synchronization is to create synchronized methods within classes. But it will not work in all cases, for example, if you want to synchronize access to object's of a class that was not designed for multithreaded programming or the class does not use synchronized methods. In this situation, the synchronized statement is a solution. Synchronized statements are similar to synchronized methods. It is used to acquire a lock on an object before performing an action.

The synchronized statement is different from a synchronized method in the sense that it can be used with the lock of any object and the synchronized method can only be used with its object's (or class's) lock. It is also different in the sense that it applies to a statement block, rather than an entire method. The syntax of the synchronized statement is as follows:

```
synchronized (object)
{
// statement(s)
}
```

The statement(s) enclosed by the braces are only executed when the current thread acquires the lock for the object or class enclosed by parentheses.

Now let us see how interthread communication takes place in java.

6.6. INTERTHREAD COMMUNICATION

Java provides a very efficient way through which multiple-threads can communicate with each-other. This way reduces the CPU idle time *i.e.*, A process where, a thread is paused running in its critical region and another thread is allowed to enter

NOTES

(or lock) in the same critical section to be executed. This technique is known as **Interthread communication** which is implemented by some methods. These methods are defined in "**java.lang**" package and can only be called within synchronized code shown as:

Method	*Description*
wait( )	It indicates the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls method **notify()** or **notifyAll ()**.
notify( )	It wakes up the first thread that called **wait()** on the same object.
notifyAll( )	Wakes up (Unlock) all the threads that called **wait ( )** on the same object. The highest priority thread will run first.

All these methods must be called within a try-catch block. Let's see an example implementing these methods:

```
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);
    }}
```

Output of the Program:

C:\nisha>javac InterThread.java

C:\nisha>java InterThread

Produce: 1

consume: 1

In this program, two threads "**Producer**" and "**Consumer**" share the **synchronized** methods of the class **"Shared"**. At time of program execution, the **"put( )"** method is invoked through the **"Producer"** class which increments the variable **"num"** by 1. After producing 1 by the producer, the method **"get( )"** is invoked by through the **"Consumer"** class which retrieves the produced number and returns it to the output. Thus the Consumer can't retrieve the number without producing of it.

NOTES

Another program demonstrates the uses of wait() & notify() methods:

```
public class DemoWait extends Thread
{  int val=20;
  public static void main(String args[])  {
  DemoWait d=new DemoWait();
  d.start();
  new Demo1(d);
  }
  public void run(){
  try  {
  synchronized(this){
  wait();
  System.out.println("value is :     "+val);
  }
  }catch(Exception e){}
}
  public void valchange(int val){
  this.val=val;
  try  {
  synchronized(this)  {
  notifyAll();
  }
  }catch(Exception e){}
  }}
class Demo1 extends Thread{
  DemoWait d;
  Demo1(DemoWait d)  {
  this.d=d;
  start();
  } public void run(){
try{ System.out.println("Demo1 value is " + d.val);
  d.valchange(40);
  }catch(Exception e){}
  }}
```

Output of the Program:

C:\j2se6\thread>javac DemoWait.java

C:\j2se6\thread>java DemoWait

Demo1 value is20

value is: 40

NOTES

SUMMARY

- Multitasking allow to execute 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. Multitasking allow processes (*i.e.*, programs) to run concurrently on the program.
- 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.
- The 'main()' method in Java is referred to the thread that is running, whenever a Java program runs. It calls the main thread because it is the first thread that starts running when a program begins.
- 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.
- Synchronized methods are used to coordinate access to objects that are shared among multiple threads. These methods are declared with the synchronized keyword. Only one synchronized method at a time can be invoked for an object at a given point of time.
- When a synchronized method is invoked for a given object, it tries to acquire the lock for that object. If it succeeds, no other synchronized method may be invoked for that object until the lock is released.

REVIEW QUESTIONS

1. What is multithreading concept in Java? Also write it's advantages over multitasking.
2. Write short notes on:
   - Multitasking.
   - The Main Thread.
   - Java thread Model.
3. What are the thread Priorities? Explain with an example.
4. Define Synchronization in Java with an example.
5. Explain how threads are created by implementing Runnable interface
6. Define the term wait(), notify() and notify All().
7. Explain with a program of the uses of wait() and notify() methods.
8. Explain with an example Interthreaded Communication.

NOTES

UNIT

7

I/O IN JAVA

STRUCTURE

7.0 Learning Objectives
7.1 I/O Basics
7.2 Streams and Stream Classes
7.3 The Predefined Streams
7.4 Reading form and Writing to Console
7.5 Reading and Writing File
7.6 The Transient and Volatile Modifiers
7.7 Using Native Methods
- *Summary*
- *Review Questions*

7.0. LEARNING OBJECTIVES

After going through this unit, you will be able to :

- explain I/O basics
- discuss transient and volatile modifiers
- describe predefined streams
- explain reading and writing file
- discuss stream and stream classes.

7.1. I/O BASICS

Here are some basic points about I/O:

- Data in files on your system is called *persistent* data because it persists after the program runs.
- Files are created through *streams* in Java code.
- A stream is a linear, sequential flow of bytes of input or output data.
- Streams are written to the file system to create files.

NOTES

- It can move the work from the server to the client, making a web solution more scalable with the number of users/clients.
- If a standalone program talks to a web server, that server normally needs to support all prior versions for users which have not kept their client software updated. In contrast, a properly configured browser loads (and caches) the latest applet version, so there is no need to support legacy versions.
- The applet naturally supports the changing user state, such as figure positions on the chessboard.
- Developers can develop and debug an applet direct simply by creating a main routine and calling init() and start() on the applet.
- An untrusted applet has no access to the local machine and can only access the server it came from. This makes such an applet much safer to run than a standalone executable that it could replace. However, a signed applet can have full access to the machine it is running on if the user agrees.
- Java applets are fast-and can even have similar performance to native installed software.

Disadvantages: A Java applet may have any of the following disadvantages:

- It requires the Java plug-in.
- Some organizations only allow software installed by the administrators. As a result, some users can only view applets that are important enough to justify contacting the administrator to request installation of the Java plug-in.
- As with any client-side scripting, security restrictions may make it difficult or even impossible for an untrusted applet to achieve the desired goals.
- Some applets require a specific JRE. This is discouraged.
- If an applet requires a newer JRE than available on the system, or a specific JRE, the user running it the first time will need to wait for the large JRE download to complete.
- Java automatic installation or update may fail if a proxy server is used to access the web. This makes applets with specific requirements impossible to run unless Java is manually updated. The Java automatic updater that is part of a Java installation also may be complex to configure if it must work through a proxy.
- Unlike the older applet tag, the object tag needs workarounds to write a cross-browser HTML document.

Below is the list given for Do's and Don'ts of Java Applets:

Do's

- Draw pictures on a web page.
- Create a new window and draw the picture in it.
- Play sounds.
- Receive input from the user through the keyboard or the mouse.
- Make a network connection to the server from where the Applet is downloaded, and send to and receive arbitrary data from that server.

Don'ts

- Write data on any of the host's disks.
- Read any data from the host's disks without the user's permission. In some environments, notably Netscape, an Applet cannot read data from the user's disks even with permission.
- Delete files.

- Read from or write to arbitrary blocks of memory, even on a non-memory-protected operating system like the MacOS
- Make a network connection to a host on the Internet other than the one from which it was downloaded.
- Call the native API directly (though Java API calls may eventually lead back to native API calls).
- Introduce a virus or Trojan horse into the host system.

8.2. THE APPLET ARCHITECTURE

Java Applets are essentially Java programs that run within a web page. Applet programs are Java classes that extend the Java. Applet. Applet classes are embedded by reference within a HTML page. You can observe that when Applets are combined with HTML, they can make an interface more dynamic and powerful than with HTML alone. While some Applets do nothing more than scroll text or play animations, but by incorporating these basic features in web pages you can make them dynamic. These dynamic web pages can be used in an enterprise application to view or manipulate data coming from some source on the server. For example, an Applet may be used to browse and modify records in a database or control runtime aspects of some other application running on the server.

Besides the class file defining the Java Applet itself, Applets can use a collection of utility classes, either by themselves or archived into a JAR file. The Applets and their class files are distributed through standard HTTP requests and therefore can be sent across firewalls with the web page data. Applet code is refreshed automatically each time the user revisits the hosting website. Therefore, keeps full application up to date on each client desktop on which it is running. Since Applets are extensions of the Java platform, you can reuse existing Java components when you build web application interface with Applets. As we'll see in example programs in this unit, we can use complex Java objects developed originally for server-side applications as components of your Applets. In fact, you can write Java code that can operate as either an Applet or an application. In Fig. 8.1, we can see that using Applet program running in a Java-enabled web browser can communicate to the server.

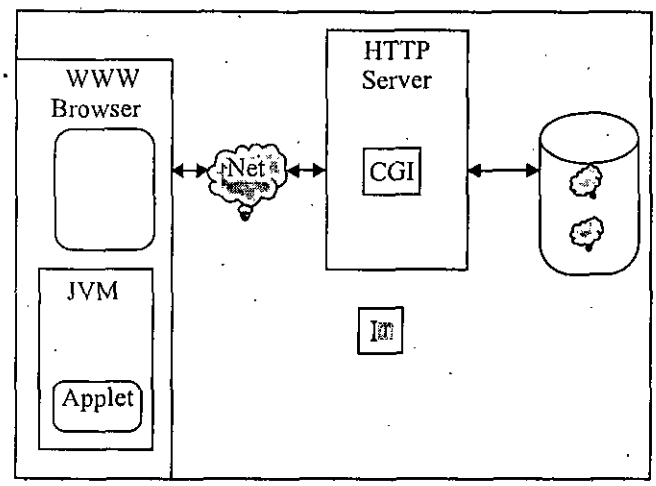


Fig. 8.1. Applet Architecture

NOTES

All applets share the same general architecture and have the same life cycle. Architecturally, applets resemble window-based, GUI programs. This means that they are not organized like console-based programs. Execution of an applet does not began at **main()**. Actually, few applets even have **main ()** methods. Instead, execution of an applet is started and controlled by its life cycle methods. Output to an applet's window is not performed by **System.out.println()** and you won't normally use a methods like **readLine()** for input. Instead, user interactions are handled by the various controls provided by AWT or Swing components, such as check boxes, lists and button. It is also possible to write output directly to an applet's windows, but you will use a method such as **drawstring ()** rather than **println ()**. Following points must be remembered in Applet Architecture:

- Applets are event driven.
- The applet waits until an event occurs.
- The AWT notifies the applet about an event by calling event handler that has been provided by t he applet. The applet takes appropriate action and then quickly return control to AWT.
- All swing components descend from the AWT container class.

8.3. AN APPLET SKELETON: INITIALIZATION AND TERMINATION

Here, we will discuss about the Applet life cycle. If you think about **the Basic Applet Life Cycle, the points listed below should be thought of:**

1. The browser reads the HTML page and finds any <APPLET> tags.
2. The browser parses the <APPLET> tag to find the CODE and possibly CODEBASE attribute.
3. The browser downloads the. **Class** file for the Applet from the URL (Uniform Resource Locator) found in the last step.
4. The browser converts the raw bytes downloaded into a Java class, that is a Java.lang.Class object.
5. The browser instantiates the Applet class to form an Applet object. This requires the Applet to have a no-args constructor.
6. The browser calls the Applet's init () method.
7. The browser calls the Applet's start () method.
8. While the Applet is running, the browser passes all the events intended for the Applet, like mouse clicks, key presses, etc. to the Applet's handle Event () method.
9. The default method paint() in Applets just draw messages or graphics (such as lines, ovals etc.) on the screen. Update events are used to tell the Applet that it needs to repaint itself.
10. The browser calls the Applet's stop () method.

NOTES

11. The browser calls the Applet's destroy () method.

In brief, we can say that, all Applets use their five following methods:

- public void init ();
- public void start();
- public void paint();
- public void stop();
- public void destroy().

Every Applet program use these methods in its life cycle. Their methods are defined in super class, **Java.applet.**

In the superclass, these are simply do-nothing methods. Subclasses may override these methods to accomplish certain tasks at certain times. For example, **public void init()** {} init () method is used to read parameters that were passed to the Applet via <PARAM> tags because it's called exactly once when the Applet starts up. If there is such need in your program you can override init() method. Since these methods are declared in the super class, the Web browser can invoke them when it needs, without knowing in advance whether the method is implemented in the super class or the subclass. This is a good example of polymorphism.

A brief description of **init (), start (), paint (), stop (), and destroy ()** methods are given below:

- **init () method:** The init() method is called *exactly once* in an Applet's life, when the Applet is first loaded. It's normally used to read PARAM tags, start downloading any other images or media files you need, and to set up the user interface. Most Applets have init () methods.
- **start () method:** The start() method is called at *least once* in an Applet's life, when the Applet is started or restarted. In some cases it may be called more than once. Many Applets you write will not have explicit start () method and will merely inherit one from their super class. A start() method is often used to start any threads the Applet will need while it runs.
- **paint () method:** The task of paint () method is to draw graphics (such as lines, rectangles, string on characters on the screen).
- **stop() method:** The stop () method is called at least once in an Applet's life, when the browser leaves the page in which the Applet is embedded. The Applet's start () method will be called if at some later point the browser returns to the page containing the Applet. In some cases the stop () method may be called multiple times in an Applet's life. Many Applets you write will not have explicit stop () methods and will merely inherit one from their super class. Your Applet should use the stop () method to pause any running threads. When your Applet is stopped, it *should* not use any CPU cycles.
- **destroy () method:** The destroy() method is called exactly once in an Applet's life, just before the web browser unloads the Applet. This method is generally used to perform any final clean-up. For example, an Applet that stores state on the server might send some data back to the server before it is terminated. Many Applet programs generally don't have explicit destroy () methods and just inherit one from their super class.

Let us take one example to explain the use of the methods explained above. For example in a video Applet, the init () method might draw the controls and start loading

NOTES

the video file. The start () method would wait until the file was loaded, and then start playing it. The stop () method would pause the video, but not rewind it. If the start () method were called again, the video would pick up from where it left off; it would not start over from the beginning. However, if destroy () were called and then init (), the video would start over from the beginning.

The point to note here is, if you run an Applet program using Appletviewer, selecting the restart menu item calls stop () and then start (). Selecting the Reload menu item calls stop (), destroy (), and init (), in the order.

Note 1: The Applet start () and stop () methods are not related to the similarly named methods in the Java.lang.

Note 2: (*i*) Your own code may occasionally invoke star t() and stop(). For example, it is customary to stop playing an animation when the user clicks the mouse in the Applet and restart it when s/he clicks the mouse again.

(*ii*) Now we are familiar with the basic needs and ways to write Applet program. You can see a simple Applet program given below to say Hello World.

```
import Java.applet.Applet;
import Java.awt.Graphics;
public class HelloWorldApplet extends Applet
{
public void paint(Graphics g)
{
g.drawString("Hello world!", 50, 25);
}
}
```

If you observe, that this Applet version of Hello World is a little more complicated than to write an application program to say Hello World, it will take a little more effort to run it as well.

First you have to type in the source code and save it into file called HelloWorldApplet.Java.

Compile this file in the usual way. If all is well a file called HelloWorldApplet.class will be created.

Now you need to create an HTML file that will include your Applet. The following simple HTML file will do.

```
<HTML>
<HEAD>
<TITLE> Hello World </TITLE>
</HEAD>
<BODY>
This is the Applet :<P>
<Applet code="HelloWorldApplet" width="150" height="50">
</Applet>
</BODY>
</HTML>
```

NOTES

Save this file as HelloWorldApplet.html in the same directory as the HelloWorldApplet.class file.

When you've done that, load the HTML file into a Java enabled browser like Internet Explorer or Sun's Applet viewer. You should see something like below, though of course the exact details depend on which browser you use.

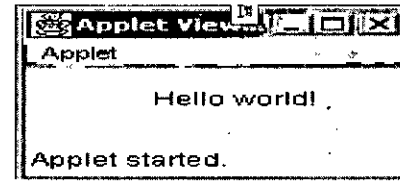
If the Applet is compiled without error and produced a HelloWorldApplet.class file, and yet you don't see the string "Hello World" in your browser chances are that the class file is in the wrong place. Make sure HelloWorldApplet.class is in the same directory as HelloWorld.html.

Also make sure that you're using a version of Netscape or Internet Explorer which supports Java. Not all versions do.

In any case Netscape's Java support is less than the perfect, so if you have trouble with an Applet, the first thing to try is load it into Sun's Applet Viewer. If the Applet Viewer has a problem, then chances are pretty good the problem is with the Applet and not with the browser.

According to Sun *"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 provides a standard interface between Applets and their environment."*

The output of the program will be like:



8.4. HANDLING EVENTS

You are leaving for work in the morning and someone rings the doorbell.... That is an event!

In life, you encounter events that force you to suspend other activities and respond to them immediately. In Java, events represent all actions that go on between the user and the application. Java's Abstract Windowing Toolkit (AWT) communicates these actions to the programs using events. When the user interacts with a program let us say by clicking a command button, the system creates an event representing the action and delegates it to the event-handling code within the program. This code determines how to handle the event so the user gets the appropriate response.

Originally, JDK 1.0.2 applications handled events using an inheritance model. A container sub class inherits the action () and handle Event () methods of its parent and handled the events for all components it contained. For instance, the following code would represent the event handler for a panel containing an OK and a Cancel button:

```
public boolean handleEvent (Java.awt.Event e)
{
if (e.id == Java.awt.Event.ACTION_EVENT)
{
```

NOTES

```
if (e.target = = buttonOK)
{
buttonOKPressed();
}
else if (e.target == buttonCancel)
{
buttonCancelPressed();
}
}
return super.handleEvent(e);
}
```

The problem with this method is that events cannot be delivered to specific objects. Instead, they have to be routed through a universal handler, which increases complexity and therefore, weakens your design.

But **Java 1.1** onward has introduced the concepts of the ***event delegation*** model. This model allows special classes, known as "adapter classes" to be built and be registered with a component in order to handle certain events. Three simple steps are required to use this model:

1. Implement the desired listener interface in your adapter class. Depending on what event you're handling, a number of listener interfaces are available. These include:

   ActionListener, WindowListener, MouseListener, MouseMotion Listener, ComponentListener, FocusListener, and ListSelection Listener.

2. Register the adapter listener with the desired component(s). This can be in the form of an add XXX Listener () method supported by the component for example include add ActionListener (), add MouseListener (), and add FocusListener ().
3. Implement the listener interface's methods in your adapter class. It is in this code that you will actually handle the event.

The event delegation model allows the developer to separate the component's display (user interface) from the event handling (application data) which results in a cleaner and more object-oriented design.

Components of an Event: Can be put under the following categories.

1. *Event Object:* When the user interacts with the application by clicking a mouse button or pressing a key an event is generated. The Operating System traps this event and the data associated with it. For example, info about time at which the event occurred, the event types (like keypress or mouse click etc.). This data is then passed on to the application to which the event belongs.

You must note that, in Java, objects, which describe the events themselves, represent events. Java has a number of classes that describe and handle different categories of events.

2. *Event Source:* An event source is the object that generated the event, for example, if you click a button an ActionEvent Object is generated. The object of the ActionEvent class contains information about the event (button click).

NOTES

3. ***Event-Handler:*** Is a method that understands the event and processes it. The event-handler method takes the Event object as a parameter. You can specify the objects that are to be notified when a specific event occurs. If the event is irrelevant, it is discarded. The four main components based on this model are ***Event classes, Event Listeners, Explicit event handling and Adapters***.

Let me give you a closer look at them one by one.

Event Classes: The EventObject class is at the top of the event class hierarchy. It belongs to the **Java.util** package. While most of the other event classes are present in Java.awt.event package.

The get Source () method of the EventObject class returns the object that initiated the event.

The getId () method returns the nature of the event. For example, if a mouse event occurs, you can find out whether the event was a click, a press, a move or release from the event object.

AWT provides two conceptual types of events: **Semantic and Low-level events.**

Semantic event: These are defined at a higher-level to encapsulate the semantics of user interface component's model.

Now let us see what the various semantic event classes are and when they are generated:

An **ActionEvent** object is generated when a component is activated.

An **Adjustment Event** Object is generated when scrollbars and other adjustment elements are used.

A **Text Event** object is generated when text of a component is modified.

An **Item Event** is generated when an item from a list, a choice or checkbox is selected. **Low-Level Events:** These events are those that represent a low-level input or windows-system occurrence on a visual component on the screen.

The various low-level event classes and what they generate are as follows:

- A **Container Event** Object is generated when components are added or removed from container.
- A **Component Event** object is generated when a component is resized moved etc.
- A **Focus Event** object is generated when component receives focus for input.
- A **Key Event** object is generated when key on keyboard is pressed, released etc.
- A **Window Event** object is generated when a window activity, like maximizing or close occurs.
- A **Mouse Event** object is generated when a mouse is used.
- A **Paint Event** object is generated when component is painted.

Event Listeners: An object delegates the task of handling an event to an **event listener**. When an event occurs, an event object of the appropriate type (as explained below) is created. This object is passed to a **Listener**. The listener must **implement the interface** that has the method for event handling. A component can have multiple listeners, and a listener can be removed using **remove Action Listener** () method. You can understand a listener as a person who has listened to your command and is doing the work which you commanded him to do so.

NOTES

As you have studied about interfaces in Block 2 Unit 3 of this course, an **Interface** contains constant values and method declaration. The Java.awt.event package contains definitions of all event classes and listener interface. The **semantic listener interfaces** defined by AWT for the above-mentioned semantic events are:

- ActionListener,
- AjdustmentListener,
- ItemListener,
- TextListener.

The **low-level event listeners** are as follows:

- ComponentListener,
- ContainerListener,
- FocusListener,
- KeyListener,
- MouseListener,
- MouseMotionListener,
- WindowsListener.

Action Event using the ActionListener interfaces: In Fig. 8.2 the Event Handling Model of AWT is given. This figure illustrates the usage of ActionEvent and ActionListener interface in a Classic Java Application.

You can see in the program given below that the Button Listener is handling the event generated by pressing button "Click Me". Each time the button "Click Me" is pressed; the message "The Button is pressed" is printed in the output area. As shown in the output of this program, the message. "The Button is pressed" is printed three times, since "Click Me" is pressed thrice.

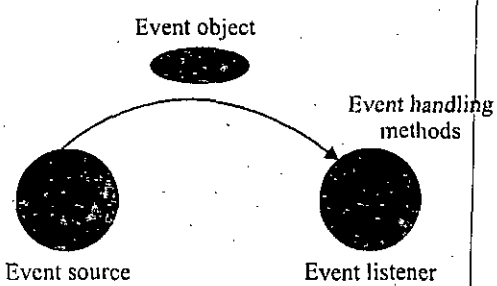


Fig. 8.2. Event Handling Model of AWT

```
import Java.awt.event.*;
import Java.awt.*;
public class MyEvent extends Frame
{
public MyEvent()
{
super("Window Title: Event Handling");
Button b1;
b1 = new Button("Click Me");
```

NOTES

```
//getContentPane().add(b1);
add(b1);
Button Listener listen = new Button Listener();
b1.add Action Listener(listen);
set Visible (true);
set Size (200,200);
}
public static void main (String args[])
{
My Event event = new My Event();
}
}; // note the semicolon
class Button Listener implements ActionListener
{
public void action Performed (ActionEvent evt)
{
Button source1 = (Button) evt. get Source ();
System. out. print ln ("The Button is Pressed");
}
}
```

Output of the program:

The Button is Pressed
The Button is Pressed
The Button is Pressed

How does the above Application work? The answer to this question is given below in steps.

1. The execution begins with the main method.
2. An Object of the MyEvent class is created in the main method, by invoking the constructor of the MyEvent class.
3. Super () method calls the constructor of the base class and sets the title of the window as given, "Windows Title: Event Handling".
4. A button object is created and placed at the center of the window.
5. Button object is added in the event.

NOTES

6. A Listener Object is created.
7. The addAction Listener () method registers the listener object for the button.
8. SetVisible () method displays the window.
9. The Application waits for the user to interact with it.
10. Then the user clicks on the button labeled "Click Me": The "ActionEvent" event is generated. Then the ActionEvent object is created and delegated to the registered listener object for processing. The Listener object contains the actionPerformed () method which processes the ActionEvent In the actionPerformed () method, the reference to the event source is retrieved using getSource () method. The message "The Button is Pressed" is printed.

8.5. HTML APPLET TAG

Till now you have written some Applets and have run them in the browser or Appletviewer. You have used basic tags needed for running an Applet program. Now you will learn some more tag that contains various attributes.

Applets are embedded in web pages using the <APPLET> and </APPLET> tags. APPLET elements accept The. *class* file with the CODE attribute. The CODE attribute tells the browser where to look for the compiled .class file. It is relative to the location of the source document.

If the Applet resides somewhere other than the same directory where the page lives on, you don't just give a URL to its location. Rather, you have to point at the CODEBASE. The CODEBASE attribute is a URL that points at the directory where the *.class* file is. The CODE attribute is the name of the *.class* file itself. For instance if on the HTML page in the previous section had you written

<APPLET CODE="HelloWorldApplet" CODEBASE="classes"
WIDTH="200" HEIGHT="200">
</APPLET>

then the browser would have tried to find HelloWorldApplet.class in the classes subdirectory inside in directory where the HTML page that included the Applet is contained. On the other hand if you had written

<APPLET CODE="HelloWorldApplet"
CODEBASE="http://www.mysite.com/classes"
WIDTH="200" HEIGHT="200">
</APPLET>

then the browser would try to retrieve the Applet from http://www.mysite.com/classes/HelloWorldApplet.class regardless of where the HTML page is residing.

If the Applet is in a non-default package, then the full package qualified name must be used. For example,

<APPLET CODE="mypackage. HelloWorldApplet"
CODEBASE="c:\Folder\Java\" WIDTH="200" HEIGHT="200">
</APPLET>

NOTES

The HEIGHT and WIDTH attributes work exactly as they do with IMG, specifying how big a rectangle the browser should set aside for the Applet. These numbers are specified in pixels.

Spacing Preferences: The <APPLET> tag has several attributes to define how it is positioned on the page.

The ALIGN attribute defines how the Applet's rectangle is placed on the page relative to other elements. Possible values include LEFT, RIGHT, TOP, TEXTTOP, MIDDLE, ABSMIDDLE, BASELINE, BOTTOM and ABSBOTTOM. This attribute is optional.

You can specify an HSPACE and a VSPACE in pixels to set the amount of blank space between an Applet and the surrounding text. The HSPACE and VSPACE attributes are optional.

```
<Applet code="HelloWorldApplet" width=200 height=200
ALIGN=RIGHT HSPACE=5 VSPACE=10>
</APPLET>
```

The ALIGN, HSPACE, and VSPACE attributes are identical to the attributes used by the <IMG> tag.

Alternate Text: The <APPLET> has an ALT attribute. An ALT attribute is used by a browser that understands the APPLET tag but for some reason cannot play the Applet. For instance, if you've turned off Java in Netscape Navigator 3.0, then the browser should display the ALT text. The ALT tag is optional.

```
<Applet code="HelloWorldApplet"
CODEBASE="c:\Folder\classes" width="200" height="200"
ALIGN="RIGHT" HSPACE="5" VSPACE="10"
ALT="Hello World!">
</APPLET>
```

ALT is not used by browsers that do not understand <APPLET> at all. For that purpose <APPLET> has been defined to require a closing tag, </APPLET>.

All raw text between the opening and closing <APPLET> tags is ignored by a Java capable browser. However, a non-Java-capable browser will ignore the <APPLET> tags instead and read the text between them. For example, the following HTML fragment says Hello World!, both with and without Java-capable browsers.

```
<Applet code="HelloWorldApplet" width=200 height=200
ALIGN=RIGHT HSPACE=5 VSPACE=10
ALT="Hello World!">
Hello World!<P>
</APPLET>
```

Naming Applets: You can give an Applet a name by using the NAME attribute of the APPLET tag. This allows communication between different Applets on the same Web page.

```
<APPLET CODE="HelloWorldApplet" NAME="Applet1"
CODEBASE="c:\Folder\Classes" WIDTH="200" HEIGHT="200"
align="right" HSPACE="5" VSPACE="10"
```

ALT="Hello World!">

Hello World!<P>

</APPLET>

NOTES

JAR Achieves: HTTP 1.0 uses a separate connection for each request. When you're downloading many small files, the time required to set up and tear down the connections can be a significant fraction of the total amount of time needed to load a page. It would be better if you could load all the HTML documents, images, Applets, and sounds in a page in one connection.

One way to do this without changing the HTTP protocol is to pack all those different files into a single archive file, perhaps a zip archive, and just download that.

We aren't quite there yet. Browsers do not yet understand archive files, but in Java 1.1 Applets do. You can pack all the images, sounds; and. class files that an Applet needs into one JAR archive and load that instead of the individual files. Applet classes do not have to be loaded directly. They can also be stored in JAR archives.

To do this you use the ARCHIVES attribute of the APPLET tag.

<APPLET CODE="HelloWorldApplet" WIDTH="200"

HEIGHT="100" ARCHIVES="HelloWorld.jar">

<hr>

Hello World!

<hr>

</APPLET>

In this example, the Applet class is still HelloWorldApplet. However, there is no HelloWorldApplet.class file to be downloaded. Instead the class is stored inside the archive file HelloWorld.jar. Sun provides a tool for creating JAR archives with its JDK 1.1 onwards.

For Example:

% jar cf HelloWorld.jar *.class

This puts all the .class files in the current directory in a file named "HelloWorld.jar". The syntax of the jar command is similar to the Unix tar command.

The Object Tag: HTML 4.0 deprecates the <APPLET> tag. Instead you are supposed to use the <OBJECT> tag. For the purposes of embedding Applets, the <OBJECT> tag is used almost exactly like the <APPLET> tag except that the class attribute becomes the classid attribute. For example,

<OBJECT classid="MyApplet" CODEBASE="c:\Folder\Classes"

width=200 height=200 ALIGN=RIGHT HSPACE=5 VSPACE=10>

</OBJECT>

The <OBJECT> tag is also used to embed ActiveX controls and other kinds of active content. It has a few additional attributes to allow it to do that. However, for the purposes of Java you don't need to know about these.

The <OBJECT> tag is supported by Netscape and Internet Explorer. It is not supported by earlier versions of these browsers. <APPLET> is unlikely to disappear anytime soon in the further.

NOTES

You can support both by placing an <APPLET> element inside an <OBJECT> element like this:

<OBJECT classid="MyApplet" width="200" height="200">

<APPLET code="MyApplet" width="200" height="200">

</APPLET>

</OBJECT>

You will notice that browsers that understand <OBJECT> will ignore its content while the browsers will display its content.

PARAM elements are the same for <OBJECT> as for <APPLET>.

Passing Parameters to Applets: Parameters are passed to Applets in NAME and VALUE attribute pairs in <PARAM> tags between the opening and closing APPLET tags. Inside the Applet, you read the values passed through the PARAM tags with the getParameter() method of the Java.Applet.Applet class.

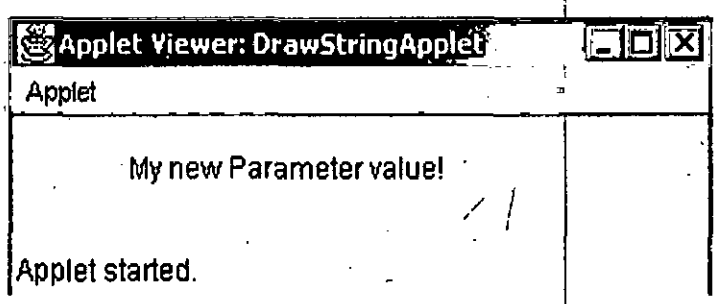
The program below demonstrates this with a generic string drawing Applet. The Applet parameter "Message" is the string to be drawn.

```
import Java.Applet.*;
import Java.awt.*;
public class DrawStringApplet extends Applet {
private String str = "Hello!";
public void paint(Graphics g) {
String str = this.getParameter("Message");
g.drawString(str, 50, 25);
}
}
```

You also need an HTML file that references your Applet. The following simple HTML file will do:

```
<HTML>
<HEAD>
<TITLE> Draw String </TITLE>
</HEAD>
<BODY>
This is the Applet:<P>
<APPLET code="Draw String Applet" width="300" height="50">
<PARAM name="Message" value="My new Parameter value!">
</APPLET>
</BODY>
</HTML>
```

NOTES



Of course you are free to change "My new Parameter value!" to a "message" of your choice. You only need to change the HTML, not the Java source code. PARAMs let you customize Applets without changing or recompiling the code.

This Applet is very similar to the HelloWorldApplet. However rather than hardcoding the message to be printed it's read into the variable str through get parameter () method from a PARAM in the HTML.

You pass get Parameter() a string that names the parameter you want. This string should match the name of a <PARAM> tag in the HTML page. getParameter() returns the value of the parameter. All values are passed as strings. If you want to get another type like an integer, then you'll need to pass it as a string and convert it to the type you really want.

The <PARAM> HTML tag is also straightforward. It occurs between <APPLET> and </APPLET>. It has two attributes of its own, NAME and VALUE. NAME identifies which PARAM this is. VALUE is the value of the PARAM as a String. Both should be enclosed in double quote marks if they contain white space.

An Applet is not limited to one PARAM. You can pass as many named PARAMs to an Applet as you like. An Applet does not necessarily need to use all the PARAMs that are in the HTML. You can be safely ignore additional PARAMs.

Processing an Unknown Number Of Parameters: Sometimes the parameters are not known to you, in that case most of the time you have a fairly good idea of what parameters will and won't be passed to your Applet or perhaps you want to write an Applet that displays several lines of text. While it would be possible to cram all this information into one long string, that's not too friendly to authors who want to use your Applet on their pages. It's much more sensible to give each line its own <PARAM> tag. If this is the case, you should name the tags via some predictable and numeric scheme. For instance in the text example the following set of <PARAM> tags would be sensible:

<PARAM name="param1" value="Hello Good Morning">

<PARAM name="param2" value="Here I am ">

SUMMARY

- An applet is just a Java class that runs in a Web Browser such as Internet Explorer or Netscape.
- To test our Applet in a Web Browser, we need to be sure that our Web Browser supports Java. Assuming that your Browser has Java turned 'on', load up the HTML file we created into the Browser. Both Internet Explorer and Netscape

NOTES

permit you to display local HTML pages by selecting File-Open from the Menu Bar and selecting the HTML page.

- Applet is a program provided by java which is designed for execution within the web browser. Applets are mostly used for a small internet and intranet applications because of small size and it's compatibility among almost all web browsers.
- The init() method is called *exactly once* in an Applet's life, when the Applet is first loaded. It's normally used to read PARAM tags, start downloading any other images or media files you need, and to set up the user interface.
- The start() method is called at *least once* in an Applet's life, when the Applet is started or restarted.
- The stop () method is called at least once in an Applet's life, when the browser leaves the page in which the Applet is embedded. The Applet's start () method will be called if at some later point the browser returns to the page containing the Applet.
- The destroy() method is called exactly once in an Applet's life, just before the web browser unloads the Applet. This method is generally used to perform any final clean-up.
- JDK 1.0.2 applications handled events using an inheritance model. A container sub class inherits the action () and handle Event () methods of its parent and handled the events for all components it contained.
- When the user interacts with the application by clicking a mouse button or pressing a key an event is generated. The Operating System traps this event and the data associated with it.
- An event source is the object that generated the event.
- Event-handler is a method that understands the event and processes it.
- An object delegates the task of handling an event to an event listener. When an event occurs, an event object of the appropriate type (as explained below) is created. This object is passed to a Listener. The listener must implement the interface that has the method for event handling.

REVIEW QUESTIONS

1. Explain what an Applet is?
2. What are the various ways to execute an Applet?
3. What are the advantages and disadvantages of Java Applets?
4. What to do and don'ts while designing a Applets?
5. Explain the Applet Architecture with a neat sketch.
6. What is the sequence of interpretation, compilation of a Java Applet?
7. How can you re-execute an Applet from Appletviewer?
8. Give in brief the description of an Applet life cycle.
9. Is it possible to access the network resources through an Applet on the browser?
10. Write a program in which whenever you click the mouse on the frame, the coordinates of the point on which the mouse is clicked are displayed on the screen?
11. Write an Applet program in which you place a button and a textarea. When you click on button, in text area Your name and address is displayed. You have to take your name and address using < PARAM>.

NOTES

UNIT

9

GRAPHICS AND USER INTERFACE

STRUCTURE

9.0 Learning Objectives
9.1 Graphics Contexts and Graphics Objects
9.2 User Interface Components
9.3 Building User Interface with AWT
9.4 Swing-Based GUI
9.5 Layouts and Layouts Manager
9.6 Container
- *Summary*
- *Review Questions*

9.0. LEARNING OBJECTIVES

After going through this unit, you will be able to:

- define graphics contexts
- explain the term graphic objects
- discuss about building user interface with AWT
- describe the layouts and layouts manager
- explain the term container.

9.1. GRAPHICS CONTEXTS AND GRAPHICS OBJECTS

We all wonder that on seeing 2D or 3D movies or graphics, even on the Internet, we see and admire good animation on various sites. The java technology provides you the platform to develop these graphics using the Graphics class. In this unit you have to overview several java capabilities for drawing two-dimensional shapes, colors and fonts. You will learn to make your own interface, which will be user-interactive and friendly. These interfaces can be made either using java AWT components or java SWING components. In this unit you will learn to use some basic components like button, text field, text area, etc.

NOTES

Interfaces using GUI allow the user to spend less time trying to remember which keystroke sequences do what and allow spend more time using the program in a productive manner. It is very important to learn how you can beautify your components placed on the canvas area using FONT and COLOR class. For placing components on different layouts knowing use of various Layout managers is essential.

How can you build your own animation using Graphics Class? A Java graphics context enables drawing on the screen. A Graphics object manages a graphics context by controlling how objects are drawn. Graphics objects contain methods for drawing, font manipulation, color manipulation and the other related operations. It has been developed using the Graphics object g (the argument to the applet's paint method) to manage the applet's graphics context. In other words you can say that Java's Graphics is capable of:

- Drawing 2D shapes,
- Controlling colors,
- Controlling fonts,
- Providing Java 2D API,
- Using More sophisticated graphics capabilities,
- Drawing custom 2D shapes,
- Filling shapes with colors and patterns.

Before we begin drawing with Graphics, you must understand Java's coordinate system, which is a scheme for identifying every possible point on the screen. Let us start with Graphics Context and Graphics Class.

Graphics Context and Graphics Class

- Enables drawing on screen,
- Graphics object manages graphics context,
- Controls how objects is drawn,
- Class Graphics is abstract,
- Cannot be instantiated,
- Contributes to Java's portability,
- Class Component method paint takes Graphics object.

The Graphics class is the abstract base class for all graphics contexts. It allows an application to draw onto components that are realized on various devices, as well as on to off-screen images.

Public Abstract Class Graphics Extends Object

You have seen that every applet performs drawing on the screen.

Graphics Objects: In Java all drawing takes place via a Graphics object. This is an instance of the class java.awt.Graphics.

Initially the Graphics object you use will be passed as an argument to an applet's paint() method. The drawing can be done Applet Panels, Frames, Buttons, Canvases etc. Each Graphics object has its own coordinate system, and methods for drawing strings, lines, rectangles, circles, polygons etc. Drawing in Java starts with particular Graphics object. You get access to the Graphics object through the paint (Graphics g) method of your applet.

Each draw method call will look like: **g.drawString("Hello World", 0, 50);**

Where g is the particular Graphics object with which you're drawing. For convenience sake in this unit the variable g will always refer to a pre-existing object of the Graphics class. It is not a rule you are free to use some other name for the particular Graphics context, such as myGraphics or applet-Graphics or anything else.

Color Control

It is known that Color enhances the appearance of a program and helps in conveying meanings. To provide color to your objects use class Color, which defines methods and constants for manipulation colors? Colors are created from **red**, **green** and **blue** components **RGB** values. All three RGB components can be integers in the range 0 to 255, or floating point values in the range 0.0 to 1.0

The first part defines the amount of red, the second defines the amount of green and the third defines the amount of blue. So, if you want to give dark red color to your graphics you will have to give the first parameter value 255 and two parameter zero.

Some of the most common colors are available by name and their RGB values in Table 9.1.

Table 9.1: Colors and their RGB values

Color Constant	*Color*	*RGB Values*
Public final static Color ORANGE	Orange	255, 200, 0
Public final static Color PINK	Pink	255, 175, 175
Public final static Color CYAN	Cyan	0, 255, 255
Public final static Color MAGENTA	Magenta	255, 0, 255
Public final static Color YELLOW	Yellow	255, 255, 0
Public final static Color BLACK	Black	0, 0, 0
Public final static Color WHITE	White	255, 255, 255
Public final static Color GRAY	Gray	128, 128, 128
Public final static Color LIGHT_GRAY	light gray	192, 192, 192
Public final static Color DARK_GRAY	dark gray	64, 64, 64
Public final static Color RED	Red	255, 0, 0
Public final static Color GREEN	Green	0, 255, 0
Public final static Color BLUE	Blue	0, 0, 255

Color Methods: To apply color in your graphical picture (objects) or text two Color methods get Color and set Color are provided. Method get Color returns a Color object representing the current drawing color and method set Color used to sets the current drawing color.

Color Constructor:

public Color(int r, int g, int b): Creates a color based on the values of red, green and blue components expressed as integers from 0 to 255.

NOTES

public Color (float r, float g, float b) : Creates a color based the values of on red, green and blue components expressed as floating-point values from 0.0 to 1.0.

Color methods:

public int getRed(): Returns a value between 0 and 255 representing the red content

public int getGreen(): Returns a value between 0 and 255 representing the green content public int getBlue(). Returns a value between 0 and 255 representing the blue content.

Graphics Methods For Manipulating Colors: public Color getColor (): Returns a Color object representing the current color for the graphics context. public void setColor (Color c): Sets the current color for drawing with the graphics context. As you do with any variable you should preferably give your colors descriptive names. For instance

Color medGray = new Color(127, 127, 127);

Color cream = new Color(255, 231, 187);

Color lightGreen = new Color (0, 55, 0);

You should note that Color is not a property of a particular rectangle, string or other object you may draw, rather color is a part of the Graphics object that does the drawing. You change the color of your Graphics object and everything you draw from that point forward will be in the new color, at least until you change it again.

When an applet starts running, its color is set to black by default. You can change this to red by calling g.setColor(Color.red). You can change it back to black by calling g.setColor(Color.black).

Fonts

You must have noticed that until now all the applets have used the default font. However unlike HTML Java allows you to choose your fonts. Java implementations are guaranteed to have a serif font like Times that can be accessed with the name "Serif", a monospaced font like courier that can be accessed with the name "Mono", and a sans serif font like Helvetica that can be accessed with the name "SansSerif".

How can you know the available fonts on your system for an applet program? You can list the fonts available on the system by using the getFontList() method from java.awt.Toolkit. This method returns an array of strings containing the names of the available fonts. These may or may not be the same as the fonts to installed on your system. It is implementation is dependent on whether or not all the fonts in a system are available to the applet.

Choosing a font face is very easy. You just create a new Font object and then call. setFont(Font f).

To instantiate a Font object the constructor

public Font(String name, int style, int size) can be used. **name** is the name of the font family, *e.g.*, "Serif", "SansSerif", or "Mono".

Size is the size of the font in points. In computer graphics a point is considered to be equal to one pixel. 12 points is a normal size font.

Style is an mnemonic constant from java.awt.Font that tells whether the text will be bold, italics or plain. The three constants are Font.PLAIN, Font.BOLD, and Font.ITALIC.

In other words, The Class Font contains methods and constants for font control. Font constructor takes three arguments

Font name Monospaced, SansSerif, Serif, etc.,

Font style Font.PLAIN, Font.ITALIC and Font.BOLD

Font size Measured in points (1/72 of inch)

Graphics method for Manipulating Fonts : public Font get Font(): Returns a Font object reference representing the current Font. public void setFont(Font f): Sets the current font, style and size specified by the Font object reference f.

Font Metrics: Sometimes you will need to know how much space a particular string will occupy. You can find this out with a FontMetrics object. FontMetrics allow you to determine the height, width or other useful characteristics of a particular string, character, or array of characters in a particular font. In Fig. 9.1 you can see a string with some of its basic characteristics.

Fig. 9.1. String Characteristics

In order to tell where and whether to wrap a String, you need to measure the string, not its length in characters, which can be variable in its width, and height in pixels. Measurements of this sort on strings clearly depend on the font that is used to draw the string. All other things being equal a 14-point string will be wider than the same string in 12 or 10-point type. To measure character and string sizes you need to look at the FontMetrics of the current font. To get a FontMetrics object for the current Graphics object you use the **java.awt.Graphics.getFontMetrics() method.**

java.awt.FontMetrics provide method stringWidth(String s) to return the width of a string in a particular font, and method getLeading() to get the appropriate line spacing for the font. There are many more methods in java.awt.FontMetrics that let you measure the height and width of specific characters as well as ascenders, descenders and more, but these three methods will be sufficient for basic programs.

Coordinate System

By Default the upper left corner of a GUI component (such as applet or window) has the coordinates (0,0). A coordinate pair is composed of *x*-coordinate (the horizontal coordinate) and a *y-coordinate (the vertical coordinate)*. The *x*-coordinate is the horizontal distance moving right from the upper left corner.

The *y*-coordinate is the vertical distance moving down from the upper left corner. The *x*-axis describes every horizontal coordinate, and the *y*-axis describes every vertical coordinate. You must note that different display cards have different resolutions (*i.e.*, the density of pixels varies). Fig. 9.2 represents coordinate system. This may cause graphics to appear to be different sizes on different monitors.

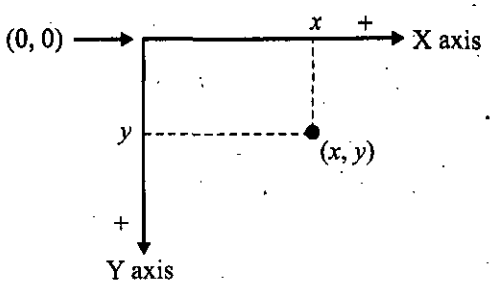


Fig. 9.2. Co-ordinate System

Now we will move towards drawing of different objects. Here, we will demonstrate drawing in applications.

Drawing Lines

Drawing straight lines with Java can be done as follows:

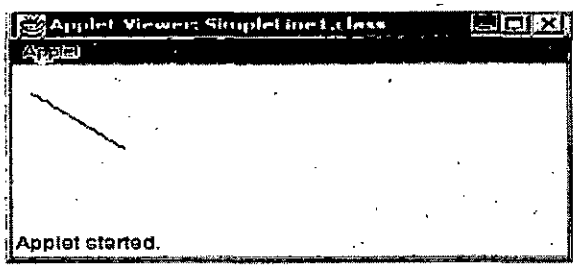
call

g.drawLine(*x1, y1, x2, y2*)

method, where *(x1, y1)* and *(x2, y2)* are the endpoints of your lines and g is the Graphics object you are drawing with. The following program will result in a line on the applet.

import java.applet.*; import java.awt.*; public class SimpleLine extends Applet { public void paint(Graphics g) { g.drawLine(10, 20, 30, 40); } }

Output:



Drawing Rectangle

Drawing rectangles is simple. Start with a Graphics object g and call its drawRect() method:

public void drawRect(int x, int y, int width, int height)

The first argument int is the left hand side of the rectangle, the second is the top of the rectangle, the third is the width and the fourth is the height. This is in contrast to some APIs where the four sides of the rectangle are given.

Remember that the upper left hand corner of the applet starts at *(0, 0)*, not at (1, 1). This means that a 100 by 200 pixel applet includes the points with *x* coordinates between 0 and 99, not between 0 and 100. Similarly the *y* coordinates are between 0 and 199 inclusive, not 0 and 200.

Drawing Ovals and Circles

Java has methods to draw outlined and filled ovals. These methods are called drawOval() and fillOval() respectively. These two methods are:

public void drawOval(int left, int top, int width, int height)

public void fillOval(int left, int top, int width, int height)

NOTES

Instead of dimensions of the oval itself, the dimensions of the smallest rectangle, which can enclose the oval, are specified. The oval is drawn as large as it can be to touch the rectangle's edges at their centers. Figure 9.3 may help you to understand properly.

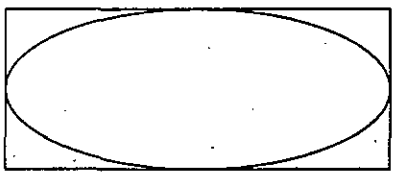


Fig. 9.3. An Oval

The arguments to **drawOval()** are the same as the arguments to **drawRect().** The first int is the left hand side of the enclosing rectangle, the second is the top of the enclosing rectangle, the third is the width and the fourth is the height. There is no special method to draw a circle. Just draw an oval inside a square.

Drawing Polygons and Polylines

You have already seen that in Java rectangles are defined by the position of their upper left hand corner, their height, and their width. However it is implicitly assumed that there is in fact an upper left hand corner. What's been assumed so far is that the sides of the rectangle are parallel to the coordinate axes. You can't yet handle a rectangle that has been rotated at an arbitrary angle.

There are some other things you can't handle either, triangles, stars, rhombuses, kites, octagons and more. To take care of this broad class of shapes Java has a **Polygon** class.

Polygons are defined by their corners. No assumptions are made about them except that they lie in a 2-D plane. The basic constructor for the Polygon class is:

public Polygon(int[] xpoints, int[] ypoints, int npoints)

xpoints is an array that contains the x coordinates of the polygon. **ypoints** is an array that contains the y coordinates. Both should have the length **npoints**. Thus to construct a right triangle with the right angle on the origin you would type

```
int[] xpoints = {0, 3, 0};
int[] ypoints = {0, 0, 4};
```

Polygon myTriangle = new Polygon(xpoints, ypoints, 3);

To draw the polygon you can use java.awt.Graphics's drawPolygon(Polygon p) method within your paint() method like this: **g.drawPolygon(myTriangle);**

You can pass the arrays and number of points directly to the drawPolygon() method if you prefer: g.drawPolygon(xpoints, ypoints, xpoints.length);

There's also an overloaded fillPolygon() method, you can call this method like

g.fillPolygon(myTriangle);

g.fillPolygon(xpoints, ypoints, xpoints.length());

To simplify user interaction and make data entry easier, Java provides different controls and interfaces. Now let us see some of the basic user interface components of Java.

9.2. USER INTERFACE COMPONENTS

Java provides many controls. Controls are components, such as buttons, labels and text boxes that can be added to containers like frames, panels and applets. The

NOTES

Java.awt package provides an integrated set of classes to manage user interface components. Components are placed on the user interface by adding them to a container. A container itself is a component. The easiest way to demonstrate interface design is by using the container you have been working with, *i.e.*, the Applet class. The simplest form of Java AWT component is the basic User Interface Component. You can create and add these to your applet without any need to know anything about creating containers or panels. In fact, your applet, even before you start painting and drawing and handling events, is an AWT container. Because an applet is a container, you can put any of AWT components, and (or) other containers, in it.

9.3. BUILDING USER INTERFACE WITH AWT

In order to add a control to a container, you need to perform the following two steps:

1. Create an object of the control by passing the required arguments to the constructor.
2. Add the component (control) to the container.

Table 9.2: (*a*) Controls in Java

Controls	*Functions*
Textbox	Accepts single line alphanumeric entry.
TextArea	Accepts multiple line alphanumeric entry.
Push button	Triggers a sequence of actions.
Label	Displays Text.
Check box	Accepts data that has a yes/no value. More than one checkbox can be selected.
Radio button	Similar to check box except that it allows the user to select a single option from a group.
Combo box	Displays a drop-down list for single item selection. It allows new value to be entered.
List box	Similar to combo box except that it allows a user to select single or multiple items. New values cannot be entered.

Table 9.2: (*b*) Classes for controls

Controls	*Class*
Textbox	TextField
TextArea	TextArea
Push button	Button
Check box	CheckBox
Radio button	CheckboxGroup with CheckBox
Combo box	Choice
List box	List

```
    Applet app = new HelloWorldApplet();
    frame.add(app);
    frame.setVisible(true);
    frame.addWindowListener(new WindowAdapter(){
    public void windowClosing(WindowEvent e){
    System.exit(0);
    }
    });
    }
    public void paint(Graphics g)
    { g.drawString("Hello World!",200,100);
    }}
```

Compiling Applets:.

javac HelloWorldApplet.java

Running Applet from console:

java HelloWorldApplet

Running Applet from Web browser:

Running applet in browser is very easy job, create an html file with the following code:

```
<html>
<head>
<title>A Simple Applet program</title>
</head>
<body>
<APPLET CODE="HelloWorldApplet.class" WIDTH=700 HEIGHT=500>
</APPLET>
</body>
</html>
```

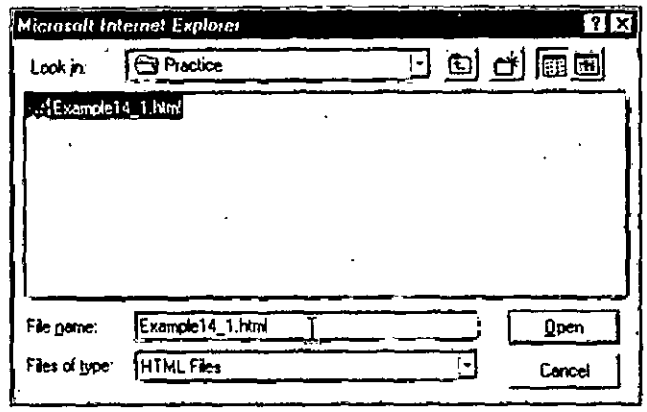
The Applet tag in html is used to embed an applet in the web page.

<APPLET CODE="*HelloWorldApplet.class*" WIDTH=700 HEIGHT=500>

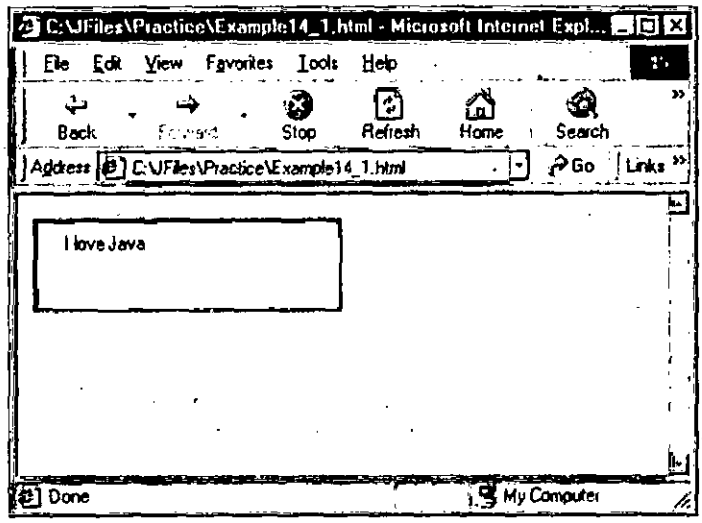
CODE tag is used to specify the name of Java applet class name. To test your applet opens the html file in web browser. You browser should display applet.

Advantages: A Java applet can have any or all of the following advantages:

- It is simple to make it work on Linux, Microsoft Windows and Mac OS X *i.e.*, to make it cross platform. Applets are supported by most web browsers.
- The same applet can work on "all" installed versions of Java at the same time, rather than just the latest plug-in version only. However, if an applet requires a later version of the Java Runtime Environment (JRE) the client will be forced to wait during the large download.
- Most web browsers cache applets so will be quick to load when returning to a web page. Applets also improve with use: after a first applet is run, the JVM is already running and starts quickly.



Here's our Applet running in Internet Explorer...



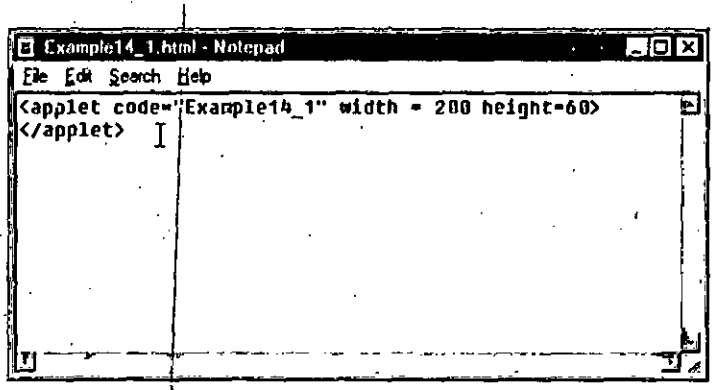
As you can see, the results are basically the same as those we found when running within the Applet Viewer—the difference is that the Web Browser is now acting as the container for our object, not the Applet Viewer.

Example of class Applet: Applet is a program provided by java which is designed for execution within the web browser. Applets are mostly used for a small internet and intranet applications because of small size and it's compatibility among almost all web browsers. Applets are also very secure. For example, Applets can be used to serve animated graphics on the web. Following example creates a simple applet that displays *Hello World message*

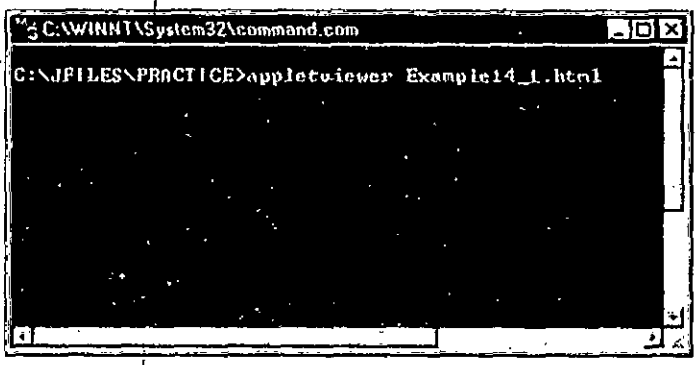
```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class HelloWorldApplet extends Applet{
  public static void main(String[] args){
  Frame frame = new Frame("Roseindia.net");
  frame.setSize(400,200);
```

NOTES

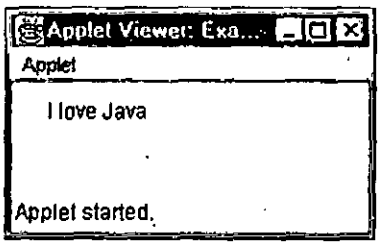
display its output in a Web Browser. You can saved the HTML as Example14_1.html HTML works primarily with tags, and the code that you see is an Applet tag to tell the Browser reading the HTML to create an instance of the Example14_1 class, within a window 200 pixels wide and 60 pixel high. We're now ready to load our Applet into a browser, but before we do that, we must be known about the Applet Viewer.



Use the Applet Viewer to test our Applet: Java includes, in its Java Developers Kit, a special interpreter called the Applet Viewer which can be used to test the Applets we write. To execute it, open up a Command Window and enter the following instructions to execute the HTML file we just wrote.



If we now press the ENTER key, we should see the Applet Viewer display window, and within it, the results of the execution of our Java applet...



The Applet Viewer can be shut down simply by clicking on its close button or Control Menu Icon.

Display our Applet in a Web Browser: To test our Applet in a Web Browser, we need to be sure that our Web Browser supports Java. Assuming that your Browser has Java turned 'on', load up the HTML file we created into the Browser. Both Internet Explorer and Netscape permit you to display local HTML pages by selecting File-Open from the Menu Bar and selecting the HTML page.

```
import java.awt.*;
import java.applet.*;
public class Example14_1 extends Applet {
  public void paint(Graphics g) {
    g.drawString("I love Java",20,20);
  }
}
```

Let's take a closer look at the code and then we'll compile it and discuss how to 'run' it in a Web page.

The first thing to note is the two import statements. We need to import 'awt' in order to produce the window within our browser. The second import statement is necessary because all Applets extend, or inherit from, the Java applet class.

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

Because of that, the line of code defining the Applet class must use the keyword 'extends' as this one does here...

public class Example14_1 extends Applet {

The second thing to note is that Java Applets, unlike the Java Applications we followed in the book, may not have a main() method. A Java Applet, when it fires up in a Web Browser, automatically looks to execute three methods in this order.

- init()
- start()
- paint

We can choose which of these to code:

- The init() method is executed when an Applet object is first instantiated.
- The start() method is executed after the init() method completes execution and every time the user of the browser returns to the HTML page on which the applet resides.
- The paint() method is called after the init method has completed and after the start() method has begun-most importantly, code in the paint() method is guaranteed to run each time the window needs to be re-drawn (*i.e.*, after the browser has been minimized, covered, the HTML page has changed, etc.,)

For our purposes, we coded the paint method, which uses a Graphics object (here defined as 'g') to do the drawing...

public void paint(Graphics g) {

The code in the paint() method is used to draw the window that will be displayed in the Web Browser. To do that we use the drawString() method of the Graphics object, supplying the text that we wish to appear in the window and it's dimensions...

g.drawString("I love Java",20,20);

As we know that Applet classes need to run within a Web Browser. Assuming you have now compiled the Java source file into a Bytecode file, we now need to create an HTML document which will create an instance of our Applet class when displayed in a Web Browser. Here's the HTML necessary to create an instance of our object and

NOTES

UNIT

8

APPLETS

STRUCTURE

8.0 Learning Objectives
8.1 The Applet Class
8.2 The Applet Architecture
8.3 An Applet Skeleton: Initialization and Termination
8.4 Handling Events
8.5 HTML Applet Tag
- *Summary*
- *Review Questions*

8.0. LEARNING OBJECTIVES

After going through this unit, you will be able to :

- explain the applet class
- define applet architecture
- discuss applet skeleton
- describe handling events.

8.1. THE APPLET CLASS

An applet is just a Java class that runs in a Web Browser such as Internet Explorer or Netscape. Creating an Applet is really easy, unfortunately, there's one problem. Neither Internet Explorer or Netscape support the latest and greatest from Java—namely its Swing Package, so if you intend to use Swing components in your Applet we need to do a little more work. But let's start with a simple Applet so that we can see what's going on. Here's the code for our first Applet which will display 'I love Java' as a message in a Web page.

NOTES

SUMMARY

- Data in files on your system is called persistent data because it persists after the program runs.
- Streams are written to the file system to create files.
- The InputStream and OutputStream are central classes in the package which are used for reading from and writing to byte streams, respectively.
- Byte streams carry integers with values that range from 0 to 255. A diversified data can be expressed in byte format, including numerical data, executable
- Character Streams are specialized type of byte streams that can handle only textual data.
- Byte streams read bytes that fall under ASCII range and character streams read Unicode characters that include characters of many international languages.
- The InputStream class defines methods for reading bytes or arrays of bytes, marking locations in the stream, skipping bytes of input, finding out the number of bytes available for reading, and resetting the current position within the stream. An input stream is automatically opened when created.
- The character streams are capable to read 16-bit characters (byte streams read 8-bit characters). Character streams are capable to translate implicitly 8-bit data to 16-bit data or vice versa.
- Native Method is a method, which is not written in Java and is outside of the JVM in a library.

REVIEW QUESTIONS

1. What are the I/O basics in Java? Also explain the difference between InputStream and OutputStream.
2. What are the Stream and Stream Classes? Explain with an example.
3. What is the methods of InputStream class and Methods of OutputStream class?
4. Define Character Stream Classes with an example.
5. Explain Predefined Stream.
6. How you can reading and writing a file?
7. Write short notes on the following:
   (*i*) Transient and Volatile Modifiers.
   (*ii*) Transient Keyword.
   (*iii*) Volatile Modifier.
   (*iv*) Using Native Methods.
8. Write a program for I/O operation using BufferedInputStream and BufferedOutputStream.
9. Write a program using FileReader and PrintWriter classes for file handling.
10. Which class may be used for reading from console?
11. Object of which class may be used for writing on console.
12. Write a program to read the output of a file and display it on console.

NOTES

7.7. USING NATIVE METHODS

You will often feel the requirement that a Java application must communicate with the environment outside of Java. This is, perhaps, the main reason for the existence of native methods. The Java implementation needs to communicate with the underlying system – such as an operating system (as Solaris or Win32, or) a Web browser, custom hardware, (such as a PDA, Set-top-device,) etc., Regardless of the underlying system, there must be a mechanism in Java to communicate with that system. Native methods provide a simple clean approach to providing this interface between Java and non-Java world without burdening the rest of the Java application with special knowledge.

Native Method is a method, which is not written in Java and is outside of the JVM in a library. This feature is not special to Java. Most languages provide some mechanism to call routines written in another language. In C++, you must use the extern "C" statement to signal that the C++ compiler is making a call to C functions.

To declare a native method in Java, a method is preceded with native modifiers much like you use the public or static modifiers, but don't define any body for the method simply place a semicolon in its place.

For example:

public native int meth();

The following class defines a variety of native methods:

public class IHaveNatives { native public void Native1(int x) ; native static public long Native2(); native synchronized private float Native3(Object o) ;

native void Native4(int[] ary) throws Exception ; }

Native methods can be static methods, thus not requiring the creation of an object (or instance of a class). This is often convenient when using native methods to access an existing C-based library. Naturally, native methods can limit their visibility with the public, private, protected, or unspecified default access.

Every other Java method modifier can be used along with native, except abstract. This is logical, because the native modifier implies that an implementation exists, and the abstract modifier insists that there is no implementation.

The Following program is a simple demonstration of native method implementation.

```
class ShowMsgBox
{
public static void main(String [] args)
{
ShowMsgBox app = new ShowMsgBox();
app.ShowMessage("Generated with Native Method");
}
private native void ShowMessage(String msg);
{
System.loadLibrary("MsgImpl");
}
}
```

NOTES

```
{
SerialDemo a = new SerialDemo("Java", "sun");
System.out.println( "Login is = " + a);
  ObjectOutputStream 0 = new
  ObjectOutputStream( new FileOutputStream("Login.out"));
0.writeObject(a);
0.close();
// Delay:
int seconds = 10;
long t = System.currentTimeMillis()+ seconds * 1000;
while(System.currentTimeMillis() < t)
;
// Now get them back:
ObjectInputStream in = new ObjectInputStream(new
FileInputStream("Login.out"));
System.out.println("Recovering object at " + new Date());
a = (SerialDemo)in.readObject();
System.out.println( "login a = " + a);
}
}
```

Output:

Login is = Logon info:
Username: Java
Date: Thu Feb 03 04:06:22 GMT+05:30 2005
Password: sun
Recovering object at Thu Feb 03 04:06:32 GMT+05:30 2005
login a = Logon info:
Username: Java
Date: Thu Feb 03 04:06:22 GMT+05:30 2005
Password: (n/a)

In the above exercise Date and Username fields are ordinary (not **transient**), and thus are automatically serialized. However, the **password** is **transient**, and so it is not stored on the disk. Also the serialization mechanism makes no attempt to recover it. The **transient** keyword is for use with **Serializable** objects only.

Volatile Modifier: The volatile modifier is used when you are working with multiple threads. The Java language allows threads that access shared variables to keep private working copies of the variables. This allows a more efficient implementation of multiple threads. These working copies need to be reconciled with the master copies in the shared (main) memory only at prescribed synchronization points, namely when objects are locked or unlocked. As a rule, to ensure that shared variables are consistently and reliably updated, a thread should ensure that it has exclusive use of such variables by obtaining a lock and conventionally enforcing mutual exclusion for those shared variables. Only variables may be volatile. Declaring them so indicates that such methods may be modified asynchronously.

NOTES

There exists, a couple of other features of a serializable class. First, it has a zero parameter constructor. When you read the object, it needs to be able to construct and allocate memory for an object, and it is going to fill in that memory from what it has read from the serial stream. The static fields, or class attributes, are not saved because they are not part of an object.

If you do not want a data attribute to be serialized, you can make it transient. That would save on the amount of storage or transmission required to transmit an object. The transient indicates that the variable is not part of the persistent state of the object and will not be saved when the object is archived. Java defines two types of modifiers **Transient** and **Volatile.** The volatile indicates that the variable is modified asynchronously by concurrently running threads.

Transient Keyword: When an object that can be serialized, you have to consider whether all the instance variables of the object will be saved or not. Sometimes you have some objects or sub objects which carry sensitive information like password. If you serialize such objects even if information (sensitive information) is private in that object if can be accessed from outside. To control this you can turn off serialization on a field- by-field basis using the transient keyword.

See the program given below to create a login object that keeps information about a login session. In case you want to store the login data, but without the password, the easiest way to do it is to implements Serializable and mark the password field as transient.

```
//Program
import Java.io.*;
import Java.util.*;
public class SerialDemo implements Serializable
{
private Date date = new Date();
private String username;
private transient String password;
SerialDemo(String name, String pwd)
{
username = name;
password = pwd;
}
public String toString()
{
String pwd = (password == null) ? "(n/a)" : password;
return "Logon info: \n " + "Username: " + username +
"\n Date: " + date + "\n Password: " + pwd;
}
public static void main(String[] args)
throws IOException, ClassNotFoundException
```

```
System.out.println("(^z to terminate)");
// read from keyboard, write to file output stream
String s2;
while ((s2 = stdin.readLine()) != null)
outFile.println(s2);
// close disk file
outFile.close();
}
}
```

Output:

Enter some text on the keyboard...

(^z to terminate)

hello students ! enjoying Java Session

^Z

Open out.txt you will find

"hello students ! enjoying Java Session" is stored in it.

7.6. THE TRANSIENT AND VOLATILE MODIFIERS

Object serialization is very important aspect of I/O programming. Now we will discuss about the serializations.

Object Serialization: It takes all the data attributes, writes them out as an object, and reads them back in as an object. For an object to be saved to a disk file it needs to be converted to a serial form. An object can be used with streams by implementing the serializable interface. The serialization is used to indicate that objects of that class can be saved and retrieved in serial form. Object serialization is quite useful when you need to use object persistence. By object persistence, the stored object continues to serve the purpose even when no Java program is running and stored information can be retrieved in a program so it can resume functioning unlike the other objects that cease to exist when object stops running.

DataOutputStreams and **DataInputStreams** are used to write each attribute out individually, and then can read them back in on the other end. But to deal with the entire object, not its individual attributes, store away an object or send it over a stream of objects. Object serialization takes the data members of an object and stores them away or retrieves them, or sends them over a stream.

ObjectInput interface is used for input, which extends the DataInput interface, and ObjectOutput interface is used for output, which extends DataOutput. You are still going to have the methods readInt(), writeInt() and so forth. ObjectInputStream, which implements ObjectInput, is going to read what ObjectOutputStream produces.

Working of object serialization: For ObjectInput and ObjectOutput interface, the class must be serializable. The serializable characteristic is assigned when a class is first defined. Your class must implement the serializable interface. This marker is an interface that says to the Java virtual machine that you want to allow this class to be serializable. You don't have to add any additional methods or anything.

NOTES

Output:

File statistics for 'input.txt'...

Number of lines = 3

Number of characters = 7.

Writing Files: You can open a file output stream to which text can be written. For this use the FileWriter class. As always, it is best to buffer the output. The following code sets up a buffered file writer stream named outFile to write text into a file named output.txt.

PrintWriter outFile = new PrintWriter(new BufferedWriter(new FileWriter("output.txt"));

The object outFile, is an object of PrintWriter class, just like System.out. If a string, s, contains some text, to be written in **"output.text".** It is written to the file as follows:

outFile.println(s);

When finished, the file is closed as: **outFile.close();**

FileWriter constructor can be used with two arguments, where the second argument is a boolean type specifying an "append" option. For example, the expression new **FileWriter("output.txt", true)** opens "output.txt" as a file output stream. If the file currently exists, subsequent output is appended to the file.

One more possibility is of opening an existing read-only file for writing. In this case, the program terminates with an "access is denied" exception. This should be caught and dealt within the program.

```
import Java.io.*;
class FileWriteDemo
{
public static void main(String[] args) throws IOException
{  // open keyboard for input
BufferedReader stdin = new BufferedReader(new
InputStreamReader(System.in));
String s = "output.txt";
// check if output file exists
File f = new File(s);
if (f.exists())
{ System.out.print("Overwrite " + s + " (y/n)? ");
  if(!stdin.readLine().toLowerCase().equals("y"))
  return;
}
// open file for output
PrintWriter outFile = new PrintWriter(new BufferedWriter
(new FileWriter(s)));
System.out.println("Enter some text on the keyboard...");
```

NOTES

```
File f = new File(fileName);
if (f.exists())
{
System.out.print("File already exists. Overwrite (y/n)? ");
if(!stdin.readLine().toLowerCase().equals("y"))
return;
}
```

See the program written below open a text file called input.txt and to count the number of lines and characters in that file.

```
//program
import Java.io.*;
public class FileOperation
{
public static void main(String[] args) throws IOException
{
// the file must be called 'input.txt'
String s = "input.txt"
File f = new File(s);
//check if file exists
if (!f.exists())
{
System.out.println("\'" + s + "\' does not exit!");
return;
}
// open disk file for input
BufferedReader inputFile = new BufferedReader(new FileReader(s));
// read lines from the disk file, compute stats
String line;
int nLines = 0;
int nCharacters = 0;
while ((line = inputFile.readLine()) != null)
{
nLines++;
nCharacters += line.length();
}
// output file statistics
System.out.println("File statistics for \'" + s + "\'...");
System.out.println("Number of lines = " + nLines);
System.out.println("Number of characters = " + nCharacters);
inputFile.close();
}
}
```

NOTES

appear on the screen. PrintWriter is different from other input/output classes as it doesn't throw an IOException. It is necessary to send check Error message, which returns true if an error has occurred. One more side effect of this method is that it flushes the stream. We can create PrintWriter object as given below.

PrintWriter pw = new PrintWriter (System.out, true)

The ReadWriteDemo program discussed earlier for reading and then displaying the content of a file. This program will give you an idea how to use FileReader and PrintWriter classes.

We may have observed that close()method is not required for objects created for standard input and output. We should have a question in mind-whether to use System.out or PrintWriter? There is nothing wrong in using System.out for sample programs but for real world applications PrintWriter is easier.

7.5. READING AND WRITING FILE

The streams are most often used for the standard input (the keyboard) and the standard output (the CRT display). Alternatively, input can arrive from a disk file using "input redirection", and output can be written to a disk file using "output redirection".

I/O redirection is convenient, but there are limitations to it. It is not possible to read data from a file using input redirection and receive user input from the keyboard at same time. Also, it is not possible to read or write multiple files using input redirection. A more flexible mechanism to read or write disk files is available in Java through its file streams. Java has two file streams-**the file reader stream and the file writer stream**. Unlike the standard I/O streams, file stream must explicitly "open" the stream before using it. Although, it is not necessary to close after operation is over, but it is a good practice to "close" the stream.

Reading Files: Let's begin with the FileReader class. As with keyboard input, it is most efficient to work through the BufferedReader class. If input is text to be read from a file, let us say "input.txt," it is opened as a file input stream as follows:

BufferedReader inputFile=new

BufferedReader(new FileReader("input.txt"));

The line above opens, input.txt as a FileReader object and passes it to the constructor of the BufferedReader class. The result is a BufferedReader object named inputFile. To read a line of text from input.txt, use the readLine() method of the BufferedReader class.

String s = inputFile.readLine();

We can see that input.txt is not being read using input redirection. It is explicitly opened as a file input stream. This means that the keyboard is still available for input. So, user can take the name of a file, instead of "hard coding". Once you are finished with the operations on file, the file stream is closed as: **inputFile.close();**

Some additional file I/O services are available through Java's File class, which supports simple operations with filenames and paths. For example, if fileName is a string containing the name of a file, the following code checks if the file exists and, if so, proceeds only if the user enters "y" to continue.

NOTES

```
try
{
BufferedInputStream buff = new BufferedInputStream(System.in);
int in = 0;
char inChar;
do
{
in = buff.read();
inChar = (char) in;
if (in != -1)
{
response.append(inChar);
}
} while ((in != 1) & (inChar != '\n'));
buff.close();
return response.toString();
}
catch (IOException e)
{
System.out.println("Exception: " + e.getMessage());
return null;
}
}
public static void main(String[] arguments)
{
System.out.print("\nWhat is your name? ");
String input = ConsoleInput.readLine();
System.out.println("\nHello, " + input);
}}
```

Output:

C:\JAVA\BIN>Java ConsoleInput
What is your name? Java Tutorial
Hello, Java Tutorial

Writing Console Output: We have to use System.out for standard Output in Java. It is mostly used for tracing the errors or for sample programs. These sources can be associated with a writer and sink objects by wrapping them in writer object. For standard output, an OutputStreamWriter object can be used, but this is often used to retain the functionality of print and println methods. In this case the appropriate writer is PrintWriter. The second argument to PrintWriter constructor requests that the output will be flushed whenever the println method is used. This avoids the need to write explicit calls to the flush method in order to cause the pending output to

NOTES

7.3. THE PREDEFINED STREAMS

Java automatically imports the **Java.lang** package. This package defines a class called **System**, which encapsulates several aspects of run-time environment. **System** also contains three predefined stream objects, **in, out, and errs.** These **objects** are declared as public and static within System. This means they can be used by other parts of your program without reference to your **System** object.

Access to standard input, standard output and standard error streams are provided via public static **System.in**, System.out and **System.err** objects. These are usually automatically associated with a user's keyboard and screen.

System.out refers to standard output stream. By default this is the console.

System.in refers to standard input, which is keyboard by default.

System.err refers to standard error stream, which also is console by default.

System.in is an object of InputStream. **System.out and System.err** are objects of **PrintStream.** These are byte streams, even though they typically are used to read and write characters from and to the console.

The predefined PrintStreams, out, and err within system class are useful for printing diagnostics when debugging Java programs and applets. The standard input stream System.in is available, for reading inputs.

7.4. READING FORM AND WRITING TO CONSOLE

Now we will discuss how we can take input from console and see the output on console.

Reading Console Input: Java takes input from console by reading from System.in. It can be associated with these sources with reader and sink objects by wrapping them in a reader object. For System.in an InputStreamReader is appropriate. This can be further wrapped in a BufferedReader as given below, if a line-based input is required.

BufferedReader br = new BufferedReader

(new InputStreamReader(System.in))

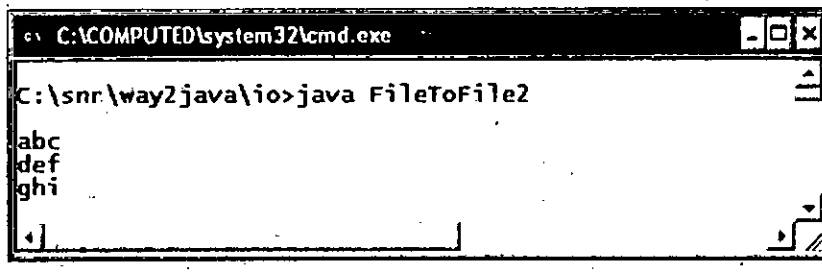
After this statement **br** is a character-based stream that is linked to the console through System.in

The program given below is for receiving console input in any of your Java applications.

```
//program
import Java.io.*;
class ConsoleInput
{
static String readLine()
{
StringBuffer response = new StringBuffer();
```

NOTES

```
        int temp;
        while( ( temp = fr.read()') != -1 )
       {        fw.write(temp);     // writes to xyz.txt
                System.out.print((char) temp);
                                    // at DOS prompt
        }  fw.close();
    fr.close(); }}
```



FileReader fr = new FileReader("pqr.txt");

FileWriter fw = new FileWriter("xyz.txt");

The explanation of this program is the same of byte streams illustrated in **FileToFile1.java**. The **pqr.txt** is the source file opened with **FileReader** constructor in **read mode**. Similarly **xyz.txt** is the destination file opened with **FileWriter** constructor in **write mode**. In case of byte sterams, the classes used are FileInputStream and FileOutputStream.

FileWriter fw = new FileWriter("xyz.txt", true);

The optional second boolean parameter **true** opens the file, "**xyz.txt**", in **append** mode.

while((temp = fr.read()) != -1)

The **read()** method of FileReader reads one byte at a time from the source file, converts into ASCII (ASCII is a subset of Unicode) integer value and returns. For example if 'A' is there in the source file, the read() method reads it and converts to 65 and returns. The same method returns –1 if EOF is encountered while reading. That is, every byte read is checked against –1 and then enters the loop.

fw.write(temp);

System.out.print((char) temp);

The **write(temp)** method of FileWriter takes the ASCII value, converts back to the original character and then writes to the destination file. To write to the DOS prompt with println() method, the ASCII value is converted to char explicitly and then written.

fw.close();

fr.close();

When the job is over, close the streams in the order of first writer stream and then reader stream.

Note: If the streams are not closed properly, sometimes buffers (maintained internally by the system) are not cleared and thereby data will not be not seen in the destination file.

Specialized Descendant Stream Classes: There are several specialized stream subclasses of the Reader and Writer class to provide additional functionality. For example, the BufferedReader not only provides buffered reading for efficiency but also provides methods such as "readLine()" to read a line from the input.

The following class hierarchy shows a few of the specialized classes in the Java.io package:

→ **Reader:**

- BufferedReader
- LineNumberReader
- FilterReader
- PushbackReader
- InputStreamReader
- FileReader
- StringReader.

→ **Writer:**

Now let us see a program to understand how the read and write methods can be used.

```
import Java.io.*;
public class ReadWriteDemo
{
public static void main(String args[]) throws Exception
{
FileReader fileread = new FileReader("StrCap.Java");
PrintWriter printwrite = new PrintWriter(System.out, true);
char c[] = new char[10];
int read = 0;
while ((read = fileread.read(c)) != -1)
printwrite.write(c, 0, read);
fileread.close();
printwrite.close();
}
}
```

File copying with Character Streams: The files can be read with character streams also. In the following program, file to file copying is done with FileReader and FileWriter. It is equivalent to **FileToFile1.java** of byte streams copying.

```
import java.io.*;
public class FileToFile2
{   public static void main(String args[]) throws IOException
    {  FileReader fr = new FileReader("pqr.txt");
       FileWriter fw = new FileWriter("xyz.txt");
```

NOTES

Character Streams are defined by using two class **Java.io.Reader and Java.io.Writer** hierarchies. Both Reader and Writer are the abstract parent classes for character-stream based classes in the Java.io package. Reader classes are used to read 16-bit character streams and Writer classes are used to write to 16-bit character streams. The methods for reading from and writing to streams found in these two classes and their descendant classes given below:

```
int read()
int read(char cbuf[])
int read(char cbuf[], int offset, int length)
int write(int c)
int write(char cbuf[])
int write(char cbuf[], int offset, int length)
```

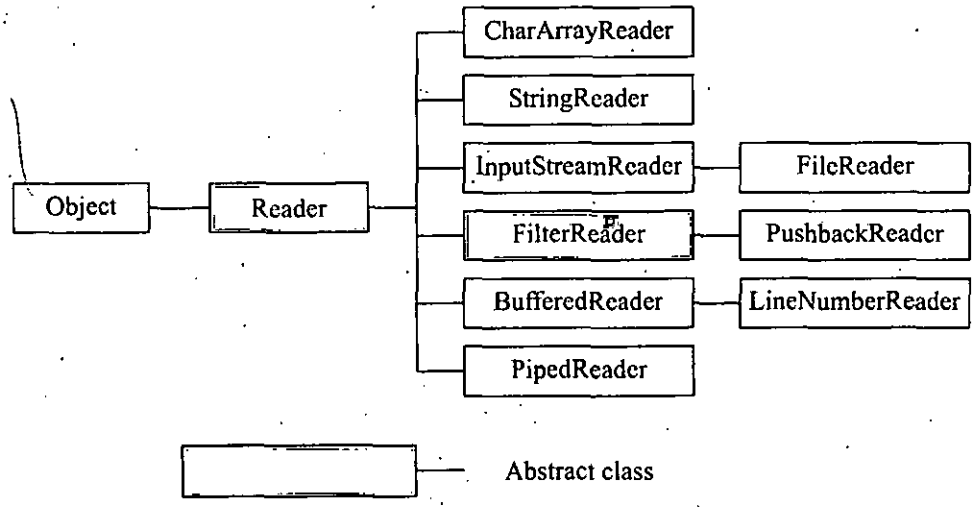


Fig. 7.3. Reader Hierarchy

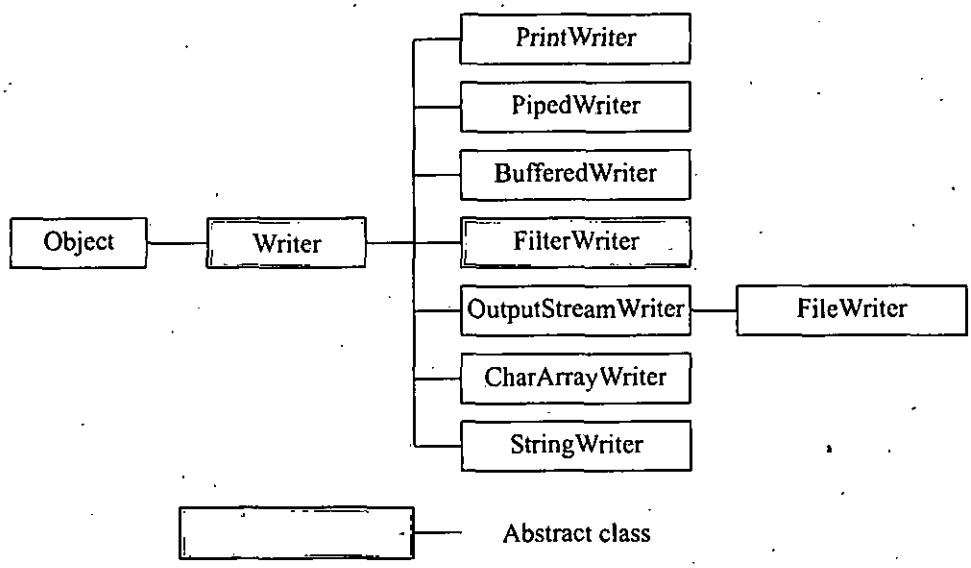


Fig. 7.4. Writer Hierarchy

```
{ buffout.write(a);
  a = a+3;
}
buffout.close();
FileInputStream filein = new FileInputStream("out.txt");
BufferedInputStream buffin = new BufferedInputStream(filein);
int i=0;
do
{  i=buffin.read();
if (i!= -1)
System.out.println(" "+ i);
} while (i != -1);
buffin.close();
}
catch (IOException e)
{ System.out.println("Error Opening a file" + e);
} ) }
```

Output:

1
4
7
10
13
16
19
22
25

Character Stream Classes

The character streams are capable to read 16-bit characters (byte streams read 8-bit characters). Character streams are capable to translate implicitly 8-bit data to 16-bit data or vice versa. The two super-most classes from which all character streams are derived are **Reader** and **Writer**. Following hierarchies can be referred for the list of all character streams.

NOTES

Filtered Streams: One of the most powerful aspects of streams is that one stream can chain to the end of another. For example, the basic input stream only provides a read()method for reading bytes. If you want to read strings and integers, attach a special data input stream to an input stream and have methods for reading strings, integers, and even floats.

The **FilterInputStream and FilterOutputStream** classes provide the capability to chain streams together. The constructors for the FilterInputStream and FilterOutputStream take InputStream and OutputStream objects as parameters:

public FilterInputStream(InputStream in)

public FilterOutputStream(OutputStream out)

FilterInputStream has four filtering subclasses, - Buffer InputStream, Data InputStream, LineNumberInputStream, and PushbackInputStream.

BufferedInputStream class: It maintains a buffer of the input data that it receives. This eliminates the need to read from the stream's source every time an input byte is needed.

DataInputStream class: It implements the DataInput interface, a set of methods that allow objects and primitive data types to be read from a stream.

LineNumberInputStream class: This is used to keep track of input line numbers.

PushbackInputStream class: It provides the capability to push data back onto the stream that it is read from so that it can be read again.

The FilterOutputStream class provides three subclasses - BufferedOutputStream, DataOutputStream and Printstream.

BufferedOutputStream class: It is the output class analogous to the BufferedInputStream class. It buffers output so that output bytes can be written to devices in larger groups.

DataOutputStream class: It implements the DataOutput interface. It provides methods that write objects and primitive data types to streams so that they can be read by the DataInput interface methods.

PrintStream class: It provides the familiar print() and println() methods.

We can see in the program given below how objects of classes FileInputStream, FileOutputStream, BufferedInputStream, and BufferedOutputStream are used for I/O operations.

```
//program
import Java.io.*;
public class StreamsIODemo
{ public static void main(String args[])
{ try
{ int a = 1;
FileOutputStream fileout = new FileOutputStream("out.txt");
BufferedOutputStream buffout = new BufferedOutputStream(fileout);
while(a<=25)
```

NOTES

ByteArrayOutputStream provides some additional methods not declared for OutputStream. The reset() method resets the output buffer to allow writing to restart at the beginning of the buffer. The write to () method is new.

- **SequenceInputStream:** Concatenate multiple input streams into one input stream.
- **StringBufferInputStream:** Allow programs to read from a StringBuffer as if it were an input stream.

Now let us see how Input and Output is being handled in the program given below: **this program creates a file and writes a string in it, and reads the number of bytes in file.**

```
// program for I/O
import Java.lang.System;
import Java.io.FileInputStream;
import Java.io.FileOutputStream;
import Java.io.File;
import Java.io.IOException;
public class FileIOOperations {
public static void main(String args[]) throws IOException {
// Create output file test.txt
FileOutputStream outStream = new FileOutputStream("test.txt");
String s = "This program is for Testing I/O Operations";
for(int i=0;i<s.length();++i)
outStream.write(s.charAt(i));
outStream.close();
// Open test.txt for input
FileInputStream inStream = new FileInputStream("test.txt");
int inBytes = inStream.available();
System.out.println("test.txt has "+inBytes+" available bytes");
byte inBuf[] = new byte[inBytes];
int bytesRead = inStream.read(inBuf,0,inBytes);
System.out.println(bytesRead+" bytes were read");
System.out.println (" Bytes read are: "+new String(inBuf));
inStream.close();
File f = new File ("test.txt");
f.delete ();
}}
```

Output:

test.txt has 42 available bytes

42 bytes were read

Bytes read are: This program is for Testing I/O Operations.

NOTES

- **InputStream Class:** The InputStream class defines methods for reading bytes or arrays of bytes, marking locations in the stream, skipping bytes of input, finding out the number of bytes available for reading, and resetting the current position within the stream. An input stream is automatically opened when created. The close() method can explicitly close a stream.

Methods of InputStream class:

- The basic method for getting data from any InputStream object is the read()method. Public abstract int read() throws IOException: reads a single byte from the input stream and returns it.
- Public int read(byte[] bytes) throws IOException: fills an array with bytes read from the stream and returns the number of bytes read.
- Public int read(byte[] bytes, int offset, int length) throws IOException: fills an array from stream starting at position offset, up to length bytes. It returns either the number of bytes read or -1 for end of file.
- Public int available() throws IOException: the readmethod always blocks when there is no data available. To avoid blocking, program might need to ask ahead of time exactly how many bytes can safely read without blocking. The available method returns this number.
- Public long skip(long n): the skip() method skips over n bytes (passed as argument of skip()method) in a stream.
- Public synchronized void mark (int readLimit): this method marks the current position in the stream so it can backed up later.

- **OutputStream class:** The OutputStream defines methods for writing bytes or arrays of bytes to the stream. An output stream is automatically opened when created. An Output stream can be explicitly closed with the close() method.

Methods of OutputStream class:

- Public abstract void write(int b) throws IOException: writes a single byte of data to an output stream.
- Public void write(byte[] bytes) throws IOException: writes the entire contents of the bytes array to the output stream.
- Public void write(byte[] bytes, int offset, int length) throws IOException: writes length number of bytes starting at position offset from the bytes array.
- The Java.io package contains several subclasses of InputStream and OutputStream that implement specific input or output functions. Some of these classes are:

  FileInputStream and FileOutputStream: Read data from or write data to a file on the native file system.

  PipedInputStream and PipedOutputStream: Implement the input and output components of a pipe. Pipes are used to channel the output from one program (or thread) into the input of another. A PipedInputStream must be connected to a PipedOutputStream and a PipedOutputStream must be connected to a PipedInputStream.

  ByteArrayInputStream and ByteArrayOutputStream: Read data from or write data to a byte array in memory.

NOTES

All the classes needed to do with reading and writing (like file copying) are placed in the package **java.io** by the designers. All the I/O streams do the file reading or writing **sequentially** (means one byte after another from start to the end of file). It can be done at **random** also. For this another class exists – **RandomAccessFile**. The **java.io** package also includes another class, **File**, to know the properties of a file like the file has read permission or write permission etc.

Most stream classes are part of the java.io package. The two main classes are **java.io.InputStream and java.io.OutputStream**. These are abstract base classes for many different subclasses with more specialized abilities, including:

- BufferedInputStream
- BufferedOutputStream
- ByteArrayInputStream
- ByteArrayOutputStream
- DataInputStream
- DataOutputStream
- FileInputStream
- FileOutputStream
- FilterInputStream
- FilterOutputStream
- LineNumberInputStream
- ObjectInputStream
- ObjectOutputStream
- PipedInputStream
- PipedOutputStream
- PrintStream
- PushbackInputStream
- SequenceInputStream
- StringBufferInputStream.

Byte Stream Classes

Java defines two major classes of byte streams: **InputStream and OutputStream**. To provide a variety of I/O capabilities subclasses are derived from these InputStream and OutputStream classes. Byte streams read bytes that fall under ASCII range and character streams read Unicode characters that include characters of many international languages. Being latest, introduced with JDK 1.1, the character streams are advantageous over byte streams like the inclusion of **newLine()** method (that is not deprecated) with BufferedReader. BufferedWriter includes a overloaded **write()** method that can write a whole string or a part of a string; but the writeBytes() method of DataOutputStream is not overloaded and writes a whole line (but not part). Sometimes, byte streams are more efficient to read binary data like the files containing images and sound. To bridge or link byte streams with character streams, there comes two classes **InputStreamReader** and **OutputStreamWriter**. Many byte streams (not all) have equivalent classes in character streams like FileInputStream equivalent FileReader and FileOutputStream equivalent to FileWriter etc.

NOTES

For example, to read files using character streams use the **Java.io.FileReader** class, and for reading it using byte streams use **Java.io.FileInputStream**.

Unless you are writing programs to work with binary data, such as image and sound files, use readers and writers (character streams) to read and write information for the following reasons:

- They can handle any character in the Unicode character set (while the byte streams are limited to ISO-Latin-1 8-bit bytes).
- They are easier to internationalize because they are not dependent upon a specific character encoding.
- They use buffering techniques internally and are therefore potentially much more efficient than byte streams.

Data can be stored on a computer system, as per the code requirements, in two ways, either **permanently** or **temporarily**. Temporary storage can be accomplished by storing the data in data structures or instance variables. The data is temporary because it is stored in RAM. For a permanent storage, the data should be stored on the hard disk either in the form of database tables or files. This tutorial is concerned with **I/O streams** used to write data to a file and later read from the file. There come two entities-a **source** and a **destination**. Source is that from where data is read and the destination is that one to where data is written. The source and destination need not be a file only; it can be a socket or keyboard input etc. To do the job of reading and writing, there come two types of streams-**input streams** and **output streams**. An input stream job is to read from the source and the output stream job is to write to the destination. That is, in the program, it is necessary to link the input stream object to the source and the output stream object to the destination.

I/O streams are carriers of data from one place to another. The input stream carries data from the source and places it temporarily in a variable (**like int k or String str etc.**) in the process (program). The output stream takes the data from the variable and writes to the destination. The variable works like a temporary buffer between input stream and output stream.

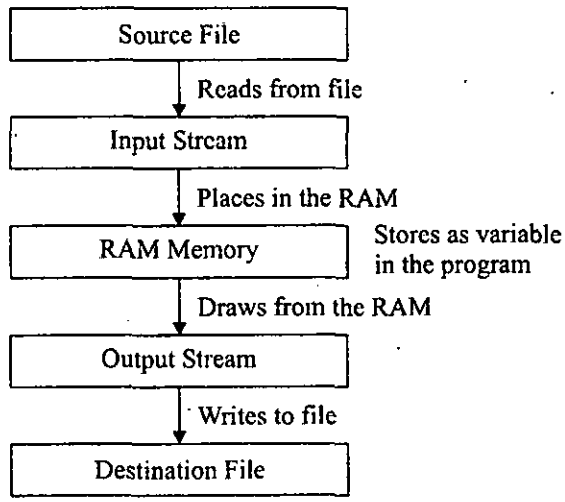


Fig. 7.2. Concept of I/O streams

It is clear from the above figure, the input stream reads from the source and puts in the buffer (actually in programming, in a temporary variable, shown later). The output stream takes from the memory and writes to the destination.

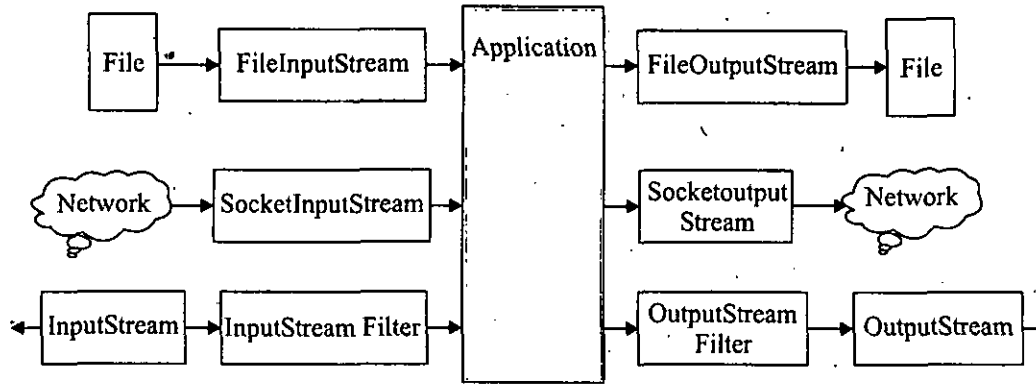


Fig. 7.1. I/O stream basics

For example:

- If you are using binary data, such as integers or doubles, then use the InputStream and OutputStream classes.
- If you are using text data, then the Reader and Writer classes are right.

Exceptions Handling during I/O: Almost every input or output method throws an exception. Therefore, any time you do an I/O operation, the program needs to catch exceptions. There is a large hierarchy of I/O exceptions derived from IOException class. Typically you can just catch IOException, which catches all the derived class exceptions. However, some exceptions thrown by I/O methods are not in the IOException hierarchy, so you should be careful about exception handling during I/O operations.

7.2. STREAMS AND STREAM CLASSES

To bring data into a program, a Java program opens a stream to a data source, such as a file or remote socket, and reads the information serially. On the flip side, a program can open a stream to a data source and write to it in a serial fashion. Whether you are reading from a file or from a socket, the concept of serially reading from, and writing to different data sources is the same. For that reason, once you understand the top level classes **(java.io.Reader, java.io.Writer),** the remaining classes are straightforward to work with. In Java IO streams are flows of data that can either read from, or write to.

The Java model for I/O is entirely based on streams. There are two types of streams: **byte streams and character streams:**

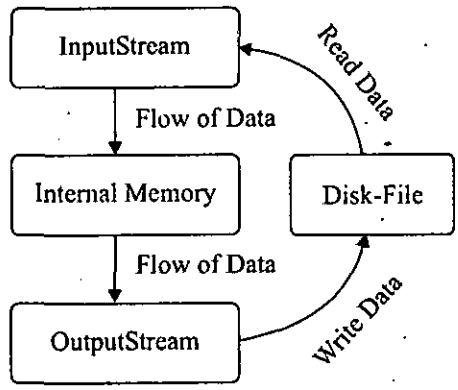
- **Byte streams** carry integers with values that range from 0 to 255. A diversified data can be expressed in byte format, including numerical data, executable programs, and byte codes-the class file that runs a Java program.
- **Character Streams** are specialized type of byte streams that can handle only textual data.

Most of the functionality available for byte streams is also provided for character streams. The methods for character streams generally accept parameters of data type char, while byte streams work with byte data types. The names of the methods in both sets of classes are almost identical except for the suffix, that is, character-stream classes end with the suffix Reader or Writer and byte-stream classes end with the suffix InputStream and OutputStream.

NOTES

disk-file may be a text file or a binary file. When we work with a text file, we use a **character** stream where one character is treated as per byte on disk. When we work with a binary file, we use a **binary** stream.

The working process of the I/O streams can be shown in the given diagram.



Java input and output are based on the use of streams, or sequences of bytes that travel from a source to a destination over a communication path. If a program is writing to a stream, you can consider it as a stream's source. If it is reading from a stream, it is the stream's destination. The communication path is dependent on the type of I/O being performed. It can consist of memory-to-memory transfers, a file system, a network, and other forms of I/O.

Streams are powerful because they abstract away the details of the communication path from input and output operations. This allows all I/O to be performed using a common set of methods. These methods can be extended to provide higher-level custom I/O capabilities.

Three streams given below are created automatically:

- System.out-standard output stream.
- System.in-standard input stream.
- System.err-standard error.

An **InputStream** represents a stream of data from which data can be read. Again, this stream will be either directly connected to a device or else to another stream.

An **OutputStream** represents a stream to which data can be written. Typically, this stream will either be directly connected to a device, such as a file or a network connection, or to another output stream.

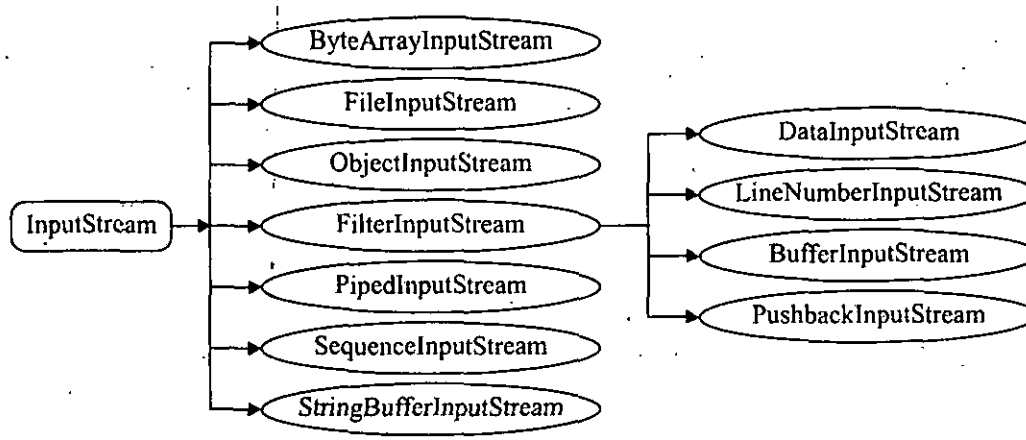
Java.io package: This package provides support for basic I/O operations. When you are dealing with the Java.io package some questions given below need to be addressed.

- What is the file format: text or binary?
- Do you want random access capability?
- Are you dealing with objects or non-objects?
- What are your sources and sinks for data?
- *Do you need to use filtering (You will know about it in later section of this unit)?*

NOTES

- **InputStream:** The **InputStream** class is used for reading the data such as a byte and array of bytes from an input source. An input source can be a **file**, a **string**, or **memory** that may contain the data. It is an abstract class that defines the programming interface for all input streams that are inherited from it. An input stream is automatically opened when you create it. You can explicitly close a stream with the **close( )** method, or let it be closed implicitly when the object is found as a garbage.

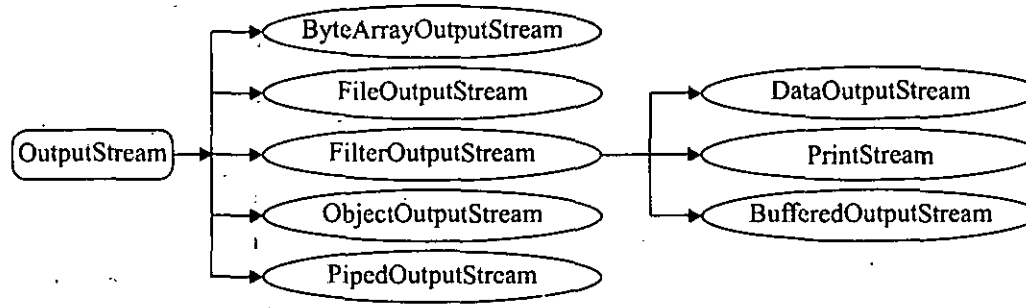
The subclasses inherited from the **InputStream** class can be seen in a hierarchy manner shown below:



InputStream is inherited from the Object class. Each class of the InputStreams provided by the java.io package is intended for a different purpose.

- **OutputStream:** The OutputStream class is a sibling to InputStream that is used for writing byte and array of bytes to an output source. Similar to input sources, an output source can be anything such as a file, a string, or memory containing the data. Like an input stream, an output stream is automatically opened when you create it. You can explicitly close an output stream with the **close( )** method, or let it be closed implicitly when the object is garbage collected.

The classes inherited from the **OutputStream** class can be seen in a hierarchy structure shown below:



OutputStream is also inherited from the Object class. Each class of the OutputStreams provided by the java.io package is intended for a different purpose.

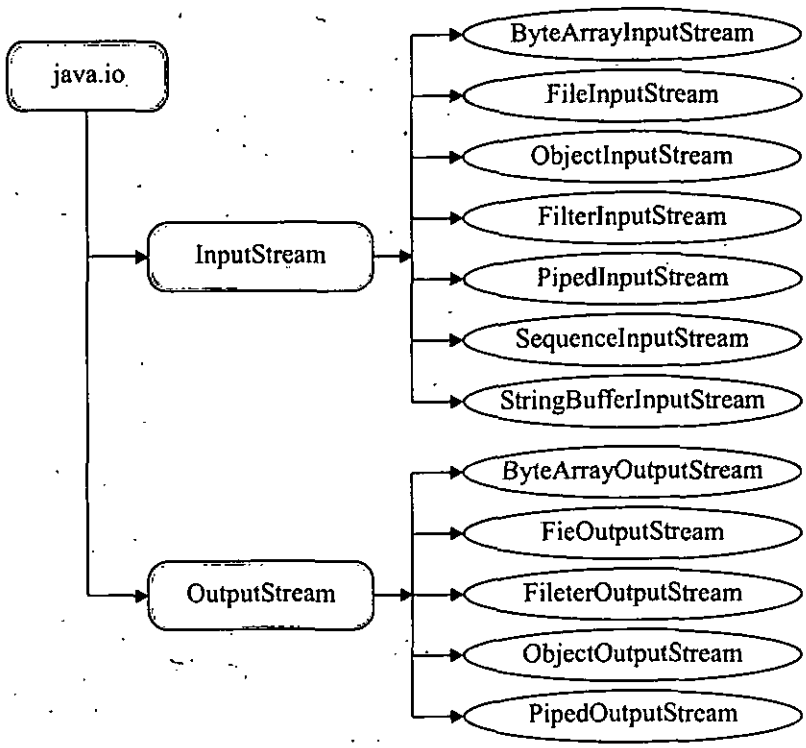
How Files and Streams Work: Java uses **streams** to handle I/O operations through which the data is flowed from one location to another. For example, an **InputStream** can flow the data from a disk file to the internal memory and an **OutputStream** can flow the data from the internal memory to a disk file. The

NOTES

- Streams can also be transferred over the Internet.
- Three streams are created for us automatically:
  Syst*em.out-standard output stream
  Syst*em.in-standard input stream
  Syst*em.err-standard error
- Input/output on the local file system using applets is dependent on the browser's security manager. Typically, I/O is not done using applets. On the other hand, stand-alone applications have no security manager by default unless the developer has added that functionality.

In fact, aside from print( ) and println( ), none of the I/O methods have been used significantly. The reason is simple: most real applications of Java are not text-based, console programs. Rather, they are graphically oriented applets that rely upon Java's Abstract Window Toolkit (AWT) for interaction with the user. Although text-based programs are excellent as teaching examples, they do not constitute an important use for Java in the real world. Also, Java's support for console I/O is limited and somewhat awkward to use—even in simple example programs. Text-based console I/O is just not very important to Java programming. The preceding paragraph notwithstanding, Java does provide strong, flexible support for I/O as it relates to files and networks. Java's I/O system is cohesive and consistent. In fact, once you understand its fundamentals, the rest of the I/O system is easy to master.

The Java Input/Output (I/O) is a part of **java.io** package. The **java.io** package contains a relatively large number of classes that support input and output operations. The classes in the package are primarily abstract classes and stream-oriented that define methods and subclasses which allow bytes to be read from and written to files or other input and output sources. The **InputStream** and **OutputStream** are central classes in the package which are used for reading from and writing to byte streams, respectively. The **java.io** package can be categories along with its stream classes in a hierarchy structure shown below:



- **The Button:** Let us first start with one of the simplest of UI components: the button. Buttons are used to trigger events in a GUI environment The Button class is used to create buttons. When you add components to the container, you don't specify a set of coordinates that indicate where the components are to be placed. A layout manager in effect for the container handles the arrangement of components. The default layout for a container is flow layout (for an applet also default layout will be flow layout). More about different layouts you will learn in later section of this unit. Now let us write a simple code to test our button class.

  To create a button use, one of the following constructors:

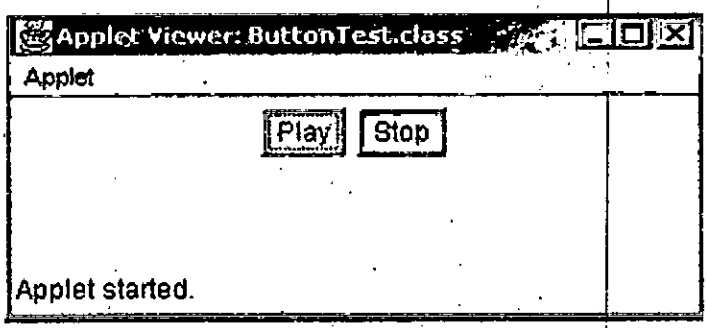
  Button() creates a button with no text label.

  Button(String) creates a button with the given string as label.

  Example Program:

```
/* <Applet code= "ButtonTest.class"
Width = 500
Height = 100>
</applet>
*/
import java.awt.*;
import java.applet.Applet;
public class ButtonTest extends Applet
{ Button b1 = new Button ("Play");
Button b2 = new Button ("Stop");
public void init()
{ add(b1);
add(b2);
} }
```

Output:



As you can see this program will place two buttons on the Applet with the caption Play and Stop.

- **The Label:** Labels are created using the Label class. Labels are basically used to identify the purpose of other components on a given interface; they cannot be

NOTES

edited directly by the user. Using a label is much easier than using a drawString( ) method because labels are drawn automatically and don't have to be handled explicitly in the paint( ) method. Labels can be laid out according to the layout manager, instead of using [x, y] coordinates, as in drawString( ).

To create a Label, use any one of the following constructors:

Label( ): creates a label with its string aligned to the left.

Label(String): creates a label initialized with the given string, and aligned left.

Label(String, int): creates a label with specified text and alignment indicated by any one of the three int arguments. Label.Right, Label.Left and Label.Center.

getText( ) method is used to indicate the current label's text setText( ) method to change the label's and text. setFont( ) method is used to change the label's font.

- **The Checkbox:** Check Boxes are labeled or unlabeled boxes that can be either "Checked off" or "Empty". Typically, they are used to select or deselect an option in a program.

  Sometimes Check are nonexclusive, which means that if you have six check boxes in a container, all the six can either be checked or unchecked at the same time. This component can be organized into Check Box Group, which is sometimes called radio buttons. Both kinds of check boxes are created using the Checkbox class. To create a nonexclusive check box you can use one of the following constructors:

  Checkbox() creates an unlabeled checkbox that is not checked.

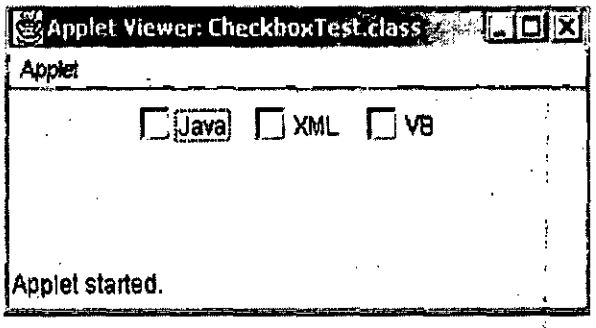
  Checkbox(String) creates an unchecked checkbox with the given label as its string.

  After you create a checkbox object, you can use the setState(boolean) method with a true value as argument for checked checkboxes, and false to get unchecked. Three checkboxes are created in the example given below, which is an applet to enable you to select up to three courses at a time.

```
import java.awt.*;
public class CheckboxTest extends java.applet.Applet
{ Checkbox c1 = new Checkbox ("Java");
Checkbox c2  = new Checkbox ("XML");
Checkbox c3  = new Checkbox ("VB");
public void init()
{ add(c1);
add(c2);
add(c3);
} }
```

NOTES

Output:



- **The Checkbox group:** CheckboxGroup is also called like a radio button or exclusive check boxes. To organize several Checkboxes into a group so that only one can be selected at a time, you can create CheckboxGroup object as follows:

  CheckboxGroup radio = new CheckboxGroup ();

  The CheckboxGroup keeps track of all the check boxes in its group. We have to use this object as an extra argument to the Checkbox constructor.

  Checkbox (String, CheckboxGroup, Boolean) creates a checkbox labeled with the given string that belongs to the CheckboxGroup indicated in the second argument. The last argument equals true if box is checked and false otherwise.

  The set Current (checkbox) method can be used to make the set of currently selected check boxes in the group. There is also a get Current () method, which returns the currently selected checkbox.

- **The Choice List:** Choice List is created from the Choice class. List has components that enable a single item to be picked from a pull-down list. We encounter this control very often on the web when filling out forms.

  The first step in creating a Choice:

You can create a choice object to hold the list, as shown below:

Choice cgender = new Choice();

Items are added to the Choice List by using addItem(String) method the object. The following code adds two items to the gender choice list.

cgender.addItem("Female");

cgender.addItem("Male");

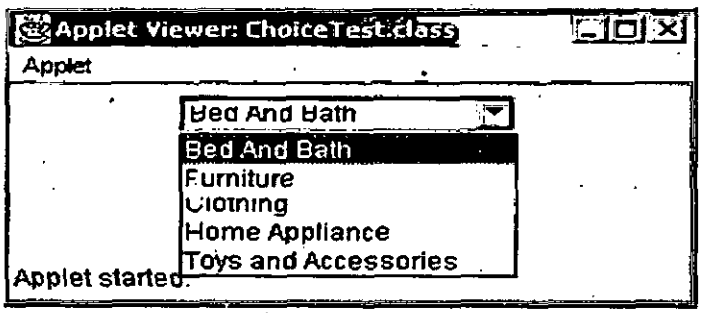
After you add the Choice List it is added to the container like any other component using the add() method.

The following example shows an Applet that contains a list of shopping items in the store.

```
import java.awt.*;
Public class ChoiceTest extends java.applet.Applet
{ Choice shoplist = new Choice();
Public void init()
{ shoplist.addItem("Bed And Bath");
shoplist.addItem("Furniture");
```

```
shoplist.addItem("Clothing");
shoplist.addItem("Home Appliance");
shoplist.addItem("Toys and Accessories");
add(shoplist); }
```

Output:



The choice list class has several methods, which are given in Table 9.3.

Table 9.3: Choice List Class Methods

Method	*Action*
getItem()	Returns the string item at the given position (items inside a choice begin at 0, just like arrays).
countItems()	Returns the number of items in the menu.
getSelectedIndex()	Returns the index position of the item that's selected.
getSelectedItem()	Returns the currently selected item as a string.
select(int)	Selects the item at the given position.
select(String)	Selects the item with the given string.

- **The Text Field:** To accept textual data from user, AWT provided two classes, TextField and TextArea. The TextField handles a single line of text and does not have scrollbars, whereas the TextArea class handles multiple lines of text. Both the classes are derived from the TextComponent class. Hence they share many common methods. TextFields provide an area where you can enter and edit a single line of text. To create a text field, use one of the following constructors:

  TextField(): creates an empty TextField with no specified width.

  TextField(int): creates an empty text field with enough width to display the specified number of characters (this has been depreciated in Java2).

  TextField(String): creates a text field initialized with the given string.

  TextField(String, int): creates a text field with specified text and specified width.

  For example, the following line creates a text field 25 characters wide with the string "Brewing Java" as its initial contents:

  TextField txtfld = new TextField ("Brewing Java", 25); add(txtfld);

NOTES

TextField, can use methods like:

setText(): Used to set the text in text field.

getText(): Used to get the text currently contained by text field.

etEditable(): Used to provide control whether the content of text field may be modified by user or not.

isEditable(): It return **true** if the text in text filed may be changed and **false** otherwise.

- **Text Area:** The TextArea is an editable text field that can handle more than one line of input. Text areas have horizontal and vertical scrollbars to scroll through the text. Adding a text area to a container is similar to adding a text field. To create a text area you can use one of the following constructors:

  TextArea(): creates an empty text area with unspecified width and height.

  TextArea(int, int): creates an empty text area with indicated number of lines and specified width in characters.

  TextArea(String): creates a text area initialized with the given string.

  TextField(String, int, int): creates a text area containing the indicated text and specified number of lines and width in the characters.

  The TextArea, similar to TextField, can use methods like setText(), getText(), setEditable(), and isEditable().

In addition, there are two more methods like these. The first is the insertText(String, int) method, used to insert indicated strings at the character index specified by the second argument. The next one is replaceText(String, int, int) method, used to replace text between given integer position specified by second and third argument with the indicated string.

The basic idea behind the AWT is that a graphical Java program is a set of nested components, starting from the outermost window all the way down to the smallest UI component. Components can include things you can actually see on the screen, such as windows, menu bars, buttons, and text fields, and they can also include containers, which in turn can contain other components.

Hope you have got a clear picture of Java AWT and its some basic UI components, In the next section of the Unit we will deal with more advance user interface components.

9.4. SWING-BASED GUI

You must be thinking that when you can make GUI interface with AWT package then what is the purpose of learning Swing-based GUI? Actually Swing has lightweight components and does not write itself to the screen, but redirects it to the component it builds on. On the other hand AWT are heavyweight and have their own view port, which sends the output to the screen. Heavyweight components also have their own *z*-ordering (look and feel) dependent on the machine on which the program is running. This is the reason why you can't combine AWT and Swing in the same container. If you do, AWT will always be drawn on top of the Swing components.

NOTES

Another difference is that Swing is pure Java, and therefore platform independent. Swing looks identically on all platforms, while AWT looks different on different platforms.

See, basically Swing provides a rich set of GUI components; features include model-UI separation and a plug able look and feel. Actually you can make your GUI also with AWT but with Swing you can make it more user-friendly and interactive. Swing components make programs efficient.

Swing GUI components are packaged into Package javax.swing.

In the Java class hierarchy there is a class:

Class Component which contains method paint for drawing Component onscreen Class Container which is a collection of related components and contains method add for adding components and Class JComponent which has Pluggable look and feel for customizing look and feel Shortcut keys (*mnemonics*)

Common event-handling capabilities, The Hierarchy is as follows:

Object⟶ Component⟶ Container⟶ Component

In Swings we have classes prefixed with the letter 'J' like

JLabel -> Displays single line of read only text

JTextField -> Displays or accepts input in a single line

JTextArea -> Displays or accepts input in multiple lines

JCheckBox ->Gives choices for multiple options

JButton -> Accepts command and does the action

JList - > Gives multiple choices and display for selection

JRadioButton - > Gives choices for multiple option, but can select one at a time.

9.5. LAYOUTS AND LAYOUTS MANAGER

When you add a component to an applet or a container, the container uses its layout manager to decide where to put the component. Different LayoutManager classes use different rules to place components.

java.awt.LayoutManager is an interface. Five classes in the java packages implement it:

- FlowLayout,
- BorderLayout,
- CardLayout,
- GridLayout,
- GridBagLayout,
- plus javax.swing.BoxLayout.
  - **FlowLayout:** A FlowLayout arranges widgets from left to right until there's no more space left. Then it begins a row lower and moves from left to right again. Each component in a FlowLayout gets as much space as it needs and no more.

    This is the default LayoutManager for applets and panels. FlowLayout is the default layout for java.awt.Panel of which java.applet.Applet is a subclasses.

NOTES

Therefore you don't need to do anything special to create a FlowLayout in an applet. However you do need to use the following constructors if you want to use a FlowLayout in a Window.LayoutManagers have constructors like any other class.

The constructor for a FlowLayout is

public FlowLayout()

Thus to create a new FlowLayout object you write

FlowLayout fl;

fl = new FlowLayout();

As usual this can be shortened to

FlowLayout fl = new FlowLayout();

You tell an applet to use a particular LayoutManager instance by passing the object to the applet's setLayout() method like this: this.setLayout(fl);

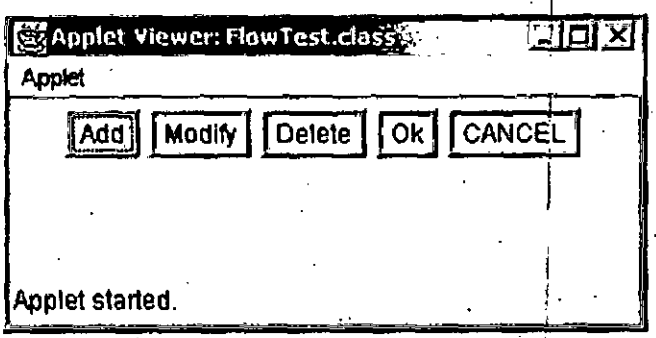
Most of the time setLayout() is called in the init() method. You normally just create the LayoutManager right inside the call to setLayout() like this

this.setLayout(new FlowLayout());

For example the following applet uses a FlowLayout to position a series of buttons that mimic the buttons on a tape deck.

```
import java.applet.*;
import java.awt.*;
public class FlowTest extends Applet {
public void init() {
this.setLayout(new FlowLayout());
this.add( new Button("Add"));
this.add( new Button("Modify"));
this.add( new Button("Delete"));
this.add( new Button("Ok"));
this.add( new Button("CANCEL"));
}
}
```

Output:



NOTES

You can change the alignment of a FlowLayout in the constructor. Components are normally centered in an applet. You can make them left or right justified. To do this just passes one of the defined constants FlowLayout.LEFT, FlowLayout.RIGHT or FlowLayout.CENTER to the constructor, *e.g.*, **this.setLayout(new FlowLayout(FlowLayout.LEFT));**

Another constructor allows you spacing option in FlowLayout:

public FlowLayout(int alignment, int horizontalSpace, int verticalSpace);

For instance to set up a FlowLayout with a ten pixel horizontal gap and a twenty pixel vertical gap, aligned with the left edge of the panel, you would use the constructor

FlowLayout fl = new FlowLayout(FlowLayout.LEFT, 20, 10);

Buttons arranged according to a center-aligned FlowLayout with a 20 pixel horizontal spacing and a 10 pixel vertical spacing

- **BorderLayout:** A BorderLayout organizes an applet into North, South, East, West and Center sections. North, South, East and West are the rectangular edges of the applet. They're continually resized to fit the sizes of the widgets included in them. Center is whatever is left over in the middle.

  A BorderLayout places objects in the North, South, East, West and center of an applet. You create a new BorderLayout object much like a FlowLayout object, in the init() method call to setLayout like this:

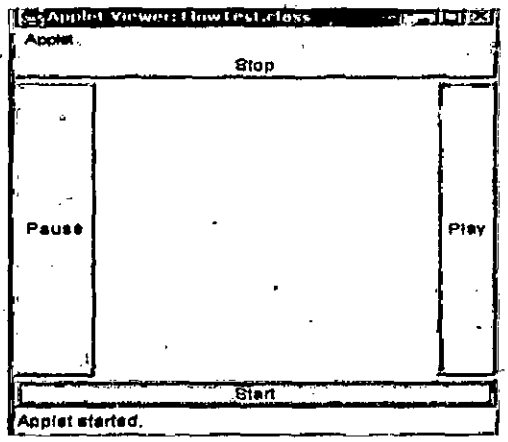
  this.setLayout(new BorderLayout());

  There's no centering, left alignment, or right alignment in a BorderLayout. However, you can add horizontal and vertical gaps between the areas. Here is how you would add a two pixel horizontal gap and a three pixel vertical gap to a BorderLayout:

  this.setLayout(new BorderLayout(2, 3));

  To add components to a BorderLayout include the name of the section you wish to add them to do like done in the program given below.

```
this.add("South", new Button("Start"));
import java.applet.*;
import java.awt.*;
public class BorderLayouttest extends Applet
{
public void init() {
this.setLayout(new BorderLayout(2, 3));
this.add("South", new Button("Start"));
this.add("North", new Button("Stop"));
this.add("East", new Button("Play"));
this.add("West", new Button("Pause"));
}
```



- **CardLayout:** A CardLayout breaks the applet into a deck of cards, each of which has its own Layout Manager. Only one card appears on the screen at a time. The user flips between cards, each of which shows a different set of components. The common analogy is with HyperCard on the Mac and Tool book on Windows. In Java this might be used for a series of data input screens, where more input is needed than will comfortably fit on a single screen.
- **GridLayout:** A GridLayout divides an applet into a specified number of rows and columns, which form a grid of cells, each equally sized and spaced. It is important to note that each is equally sized and spaced as there is another similar named Layout known as GridBagLayout .As Components are added to the layout they are placed in the cells, starting at the upper left hand corner and moving to the right and down the page. Each component is sized to fit into its cell. This tends to squeeze and stretch components unnecessarily.

You will find the GridLayout is great for arranging Panels. A GridLayout specifiesthe number of rows and columns into which components will be placed. The applet is broken up into a table of equal sized cells.

GridLayout is useful when you want to place a number of similarly sized objects. It is great for putting together lists of checkboxes and radio buttons as you did in the Ingredients applet. GridLayout looks like Fig. 9.4.

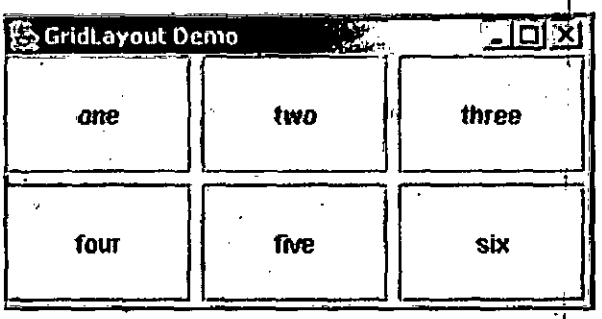


Fig. 9.4. GridLayout Demo

- **GridBagLayout:** GridBagLayout is the most precise of the five AWT Layout Managers. It is similar to the GridLayout, but components do not need to be of the same size. Each component can occupy one or more cells of the layout. Furthermore, components are not necessarily placed in the cells beginning at the upper left-hand corner and moving to the right and down.

NOTES

In simple applets with just a few components you often need only one layout manager. In more complicated applets, however, you will often split your applet into panels, lay out the panels according to a layout manager, and give each panel its own layout manager that arranges the components inside it.

The GridBagLayout constructor is trivial, GridBagLayout() with no arguments.

GridBagLayout gbl = new GridBagLayout();

Unlike the GridLayout() constructor, this does not say how many rows or columns there will be. The cells your program refers to determine this. If you put a component in row 8 and column 2, then Java will make sure there are at least nine rows and three columns. (Rows and columns start counting at zero.) If you later put a component in row 10 and column 4, Java will add the necessary extra rows and columns. You may have a picture in your mind of the finished grid, but Java does not need to know this when you create a GridBagLayout.

Unlike most other LayoutManagers you should not create a GridBagLayout inside a cell to setLayout(). You will need access to the GridBagLayout object later in the applet when you use a GridBagConstraints.

A **GridBagConstraints** object specifies the location and area of the component's display area within the container (normally the applet panel) and how the component is laid out inside its display area. The GridBagConstraints, in conjunction with the component's minimum size and the preferred size of the component's container, determines where the display area is placed within the applet.

The GridBagConstraints() constructor is trivial

GridBagConstraints gbc = new GridBagConstraints();

Your interaction with a GridBagConstraints object takes place through its eleven fields and fifteen mnemonic constants.

- **gridx and gridy:** The gridx and gridy fields specify the x and y coordinates of the cell at the upper left of the Component's display area. The upper-left-most cell has coordinates (0, 0). The mnemonic constant GridBagConstraints.RELATIVE specifies that the Component is placed immediately to the right of (gridx) or immediately below (gridy) the previous Component added to this container.
- **Gridwidth and Gridheight:** The gridwidth and gridheight fields specify the number of cells in a row (gridwidth) or column (gridheight) in the Component's display area. The mnemonic constant GridBagConstraints.REMAINDER specifies that the Component should use all remaining cells in its row (for gridwidth) or column (for gridheight). The mnemonic constant GridBagConstraints.RELATIVE specifies that the Component should fill all but the last cell in its row (gridwidth) or column (gridheight).
- **Fill:** The GridBagConstraints fill field determines whether and how a component is resized if the component's display area is larger than the component itself. The mnemonic constants you use to set this variable are

  GridBagConstraints.NONE:Don't resize the component

  GridBagConstraints.HORIZONTAL: Make the component wide enough to fill the display area, but don't change its height.

  GridBagConstraints.VERTICAL: Make the component tall enough to fill its display area, but don't change its width.

NOTES

GridBagConstraints.BOTH: Resize the component enough to completely fill its display area both vertically and horizontally.

- **Ipadx and Ipady:** Each component has a minimum width and a minimum height, smaller than which it will not be. If the component's minimum size is smaller than the component's display area, then only part of the component will be shown.

  The ipadx and ipady fields let you increase this minimum size by padding the edges of the component with extra pixels. For instance setting ipadx to two will guarantee that the component is at least four pixels wider than its normal minimum. (ipadx adds two pixels to each side.).

- **Insets:** The insets field is an instance of the java.awt.Insets class. It specifies the padding between the component and the edges of its display area.

  Anchor: When a component is smaller than its display area, the anchor field specifies where to place it in the grid cell. The mnemonic constants you use for this purpose are similar to those used in a BorderLayout but a little more specific. They are

  GridBagConstraints.CENTER

  GridBagConstraints.NORTH

  GridBagConstraints.NORTHEAST

  GridBagConstraints.EAST

  GridBagConstraints.SOUTHEAST

  GridBagConstraints.SOUTH

  GridBagConstraints.SOUTHWEST

  GridBagConstraints.WEST

  GridBagConstraints.NORTHWEST

  The default is GridBagConstraints.CENTER.

- **Weightx and weighty:** The weightx and weighty fields determine how the cells are distributed in the container when the total size of the cells is less than the size of the container. With weights of zero (the default) the cells all have the minimum size they need, and everything clumps together in the center. All the extra space is pushed to the edges of the container. It doesn't matter where they go and the default of center is fine.

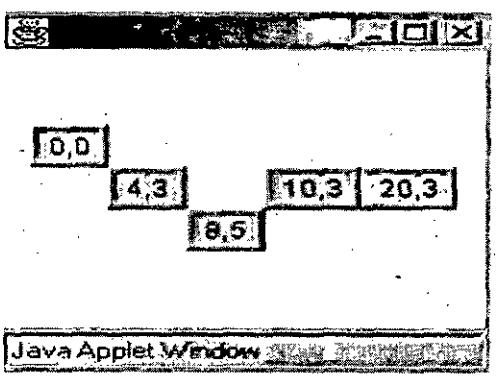
See the program given bellow for visualizing GridbagLayout.

```
import java.awt.*;
public class GridbagLayouttest extends Frame
{
Button b1,b2,b3,b4,b5;
GridBagLayout gbl=new GridBagLayout();
GridBagConstraints gbc=new GridBagConstraints();
public GridbagLayouttest()
{
setLayout(gbl);
```

NOTES

```
gbc.gridx=0;
gbc.gridy=0;
gbl.setConstraints(b1=new Button("0,0"),gbc);
gbc.gridx=4; //4th column
gbc.gridy=3; //3rd row
gbl.setConstraints(b2=new Button("4,3"),gbc);
gbc.gridx=8; //8th column
gbc.gridy=5; //5rd row
gbl.setConstraints(b3=new Button("8,5"),gbc);
gbc.gridx=10; //10th column
gbc.gridy=3; //3rd row
gbl.setConstraints(b4=new Button("10,3"),gbc);
gbc.gridx=20; //20th column
gbc.gridy=3; //3rd row
gbl.setConstraints(b5=new Button("20,3"),gbc);
add(b1);
add(b2);
add(b3);
add(b4);
add(b5);
setSize(200,200);
setVisible(true);
}
public static void main(String a[])
{
GridbagLayouttest gb= new GridbagLayouttest();
}
  }
```

Output:



NOTES

9.6. CONTAINER

You must be thinking that container will be a thing that contains something, like a bowl. Then you are right!! Actually container object is derived from the java.awt.Container class is one of (or inherited from) three primary classes: java.awt.Window, java.awt.Panel, java.awt.ScrollPane.

The Window class represents a standalone window (either an application window in the form of a java.awt.Frame, or a dialog box in the form of a java.awt.Dialog).

The java.awt.Panel class is not a standalone window by itself; instead, it acts as a background container for all other components on a form. For instance, the java.awt.Applet class is a direct descendant of java.awt.Panel.

The three steps common for all Java GUI applications are:

1. Creation of a container.
2. Layout of GUI components.
3. Handling of events.

The Container class contains the setLayout() method so that you can set the default LayoutManager to be used by your GUI. To actually add components to the container, you can use the container's add() method:

```
Panel p = new java.awt.Panel();
Button b = new java.awt.Button("OK");
p.add(b);
```

A JPanel is a Container, which means that it can contain other components. GUI design in Java relies on a layered approach where each layer uses an appropriate layout manager.

FlowLayout is the default for JPanel objects. To use a different manager use either of the following:

```
JPanel pane2 = new JPanel() // make the panel first
pane2.setLayout(new BorderLayout()); // then reset its manager
JPanel pane3 = new JPanel(new BorderLayout()); // all in one!
```

SUMMARY

- Interfaces using GUI allow the user to spend less time trying to remember which keystroke sequences do what and allow spend more time using the program in a productive manner. It is very important to learn how you can beautify your components placed on the canvas area using FONT and COLOR class.
- Controls are components, such as buttons, labels and text boxes that can be added to containers like frames, panels and applets. The Java.awt package provides an integrated set of classes to manage user interface components.
- Buttons are used to trigger events in a GUI environment
- Labels are basically used to identify the purpose of other components on a given interface; they cannot be edited directly by the user.

NOTES

- The TextField handles a single line of text and does not have scrollbars, whereas the TextArea class handles multiple lines of text. Both the classes are derived from the TextComponent class.
- TextField() creates an empty TextField with no specified width.
- A FlowLayout arranges widgets from left to right until there's no more space left.
- A BorderLayout organizes an applet into North, South, East, West and Center sections. North, South, East and West are the rectangular edges of the applet. They're continually resized to fit the sizes of the widgets included in them.
- A CardLayout breaks the applet into a deck of cards, each of which has its own Layout Manager.
- A GridLayout divides an applet into a specified number of rows and columns, which form a grid of cells, each equally sized and spaced.
- GridBagLayout is the most precise of the five AWT Layout Managers. It is similar to the GridLayout, but components do not need to be of the same size.
- A GridBagConstraints object specifies the location and area of the component's display area within the container (normally the applet panel) and how the component is laid out inside its display area.
- The gridx and gridy fields specify the x and y coordinates of the cell at the upper left of the Component's display area.
- The gridwidth and gridheight fields specify the number of cells in a row (gridwidth) or column (gridheight) in the Component's display area.
- The GridBagConstraints fill field determines whether and how a component is resized if the component's display area is larger than the component itself.
- The Window class represents a standalone window (either an application window in the form of a java.awt.Frame, or a dialog box in the form of a java.awt.Dialog).

REVIEW QUESTIONS

1. What are the Graphics contexts and Graphics Objects?
2. Write short notes on the following:
   (*i*) Color control.
   (*ii*) Fonts.
   (*iii*) Coordinate System.
   (*iv*) User Interface Components.
3. What are the various color constructors?
4. Write a program to set the Color of a String to red?
5. What is the method to retrieve the color of the text? Write a program to retrieve RGB values in a given color.
6. Write a program to set the font of your String as font name as "Arial",font size as 12 and font style as FONT.ITALIC.

NOTES

7. What is the method to retrieve the font of the text? Write a program for font retrieval.
8. Write a program that will give you the Fontmetrics parameters of a String.
9. Write a program which draws a line, a rectangle, and an oval on the applet.
10. Write a program that draws a color-filled line, a color-filled rectangle, and a color filled oval on the applet.
11. Write a program to add various checkboxes under the CheckboxGroup
12. Write a program in which the Applet displays a text area that is filled with a string, when the programs begin running.
13. Describe the features of the Swing components that subclass J Component.
14. What are the difference between Swing and AWT?
15. Why do you think Layout Manager is important?
16. How does repaint() method work with Applet?
17. How many Listeners are there for trapping mouse movements.