Everything you wanted to know about.....

Core Java

Munishwar Gulati
Mini Gulati

Learn
all techniques needed to develop applications with Java

Apply
easy steps for fast results!

Build
foundation of Programming knowledge step-by-step

Master
Programming Techniques using Java Languagae

Silicon Media™ PRESS

Core Java

Munishwar Gulati
Mini Gulati

SILICON MEDIA PRESS
New Delhi

I S B N - 8 1 - 8 7 8 7 0 - 13 - 3
First Edition 2001

Published by Silicon Media Press, Regd. Off. I-19, Lajpat Nagar - II, New Delhi. Works : KJ-75, Kavi Nagar, Ghaziabad. sm@siliconmedia.org. Ph - 4702867, 9810626977 **Typeset by** : Ram Ganga Computers (P) Ltd. and **Printed by** : Siddharth Printers, D-30, Sector-6, Noida. Ph 4444598, 9810236378.

Dedicated to our lovely daughter Priyasha

Contents

Acknowledgments

We would like to thank all of the many people who helped bring this book to print. It's really fascinating for us to write a book, because what begins as a few thoughts ends up as a physical, tangible thing that goes into the hands of many readers. The trip from thought to thing is a long one and involves inputs from lots of people. Thanks goes to all of our editors, art personnel and staff at Silicon Media.

Munishwar Gulati

I would especially want to thank my wife and co-author Mini Gulati, who always cooperated and went through the whole book to rectify me at many places.

Mini Gulati

It has been delight working with my husband and author Munishwar Gulati. He persuaded me to write, what I knew and rectified it, to give it a shape of chapter. My deep appreciation goes to him for his constant enthusiasm and hard working.

About this book

As you read the book, you will find the answers to many of the questions you have about Java Language.

How the book is organised

Chapter I introduces you to history of Java, advantage of using Java and various applications included in Java Development Kit.

Chapter 2 covers the Java basics for developing applications and applets, various datatypes, control structure and looping.

Chapter 3 explains you the concept and terms of Object Oriented Programming.

As you move on to **Chapter 4**, you will know more about thread and multithreading in java.

Chapter 5 covers all about Exception Handling.

Chapter 6 tells you about the creating Java Applets and running the applets using HTML code.

Chapter 7 introduces you to various important standard packages. **Chapter 8**, **Chapter 9** and **Chapter 10** introduces you to Language Package, Utility Package and I/O package respectively. Various important classes in thesepackages has been discussed in these packages.

Chapter 11 introduces you to Graphics fundamentals and packages used in Java.

Chapter 12 introduces you to various interactive interface elements.

This book is a sincere effort for explaining the concepts of Java Language. We sincerely hope that you find this work to be informative and enjoyable.

As a reader, you are the most important critic and commentator of our books. We value your opinion and want to know what we are doing

right, what we could do better, what areas you would like to see us publish in, and any other words of wisdom you are willing to pass.

As the publishing manager of the group that created this book, We welcome your comments. You can e-mail at siliconmedia@hotmail.com

Authors

INTRODUCTION

WHAT IS JAVA?
THE ORIGINS OF JAVA
WHY PROGRAM IN JAVA?
JAVA DEVELOPMENT KIT
PROGRAM TYPES IN JAVA

Introduction

WHAT IS JAVA?

Java is an object oriented programming language, a runtime system, a set of development tools, and an Application Programming Interface (API) developed by Sun Microsystems. Java has brought dynamic, interactive content to the Web. Java is a platform independent programming language, thereby fulfilling the programmer's dream of being able to write a program in single language, that will be compatible with all other platforms without specialized tailoring or even recompilation.

Java programs are created as a text file with the file extension .java. Java programs are compiled using the Java compiler which results in the formation of compiled bytecode with the extension .class. A machine code must run on the computer system it was compiled for, whereas bytecodes can run on any computer system equipped to handle Java programs. Bytecode is in a form that can be executed on the Java virtual machine, the core of the Java runtime system, which makes your application compatible to the client's machine, thereby enabling your program to run on any machine in the internet. A virtual machine can be considered as a microprocessor that is implemented in software and runs using the capabilities provided by your operating system and computer hardware.

The relationships between these elements are depicted in Figure 1.1.

In Java, you write programs using predefined software package of Java API. Then you compile these programs, using Java Compiler, to generate Bytecode. You can execute this bytecode using the Java Virtual Machine - The core of Java Runtime System. The Basic function of the Java Virtual Machine is to interpret the Java Bytecode and then execute it directly in the native machine instructions of the host computer. The Java Runtime System also consists of additional softwares, such as dynamic link libraries, which you need to implement the Java API on the Operating System and hardware.

The Java API has numerous platform independent software packages and it provides a single common API across all operating systems to which Java is ported. The platform independence makes Java highly portable. The keys to Java's portability are its runtime system and its API. The runtime program of Java is independent of any existing microprocessor or hardware architecture thus making it architecture natural. The simple, efficient, compact, and architecture-neutral nature of the runtime system allows it to be highly portable and still provide effective performance.

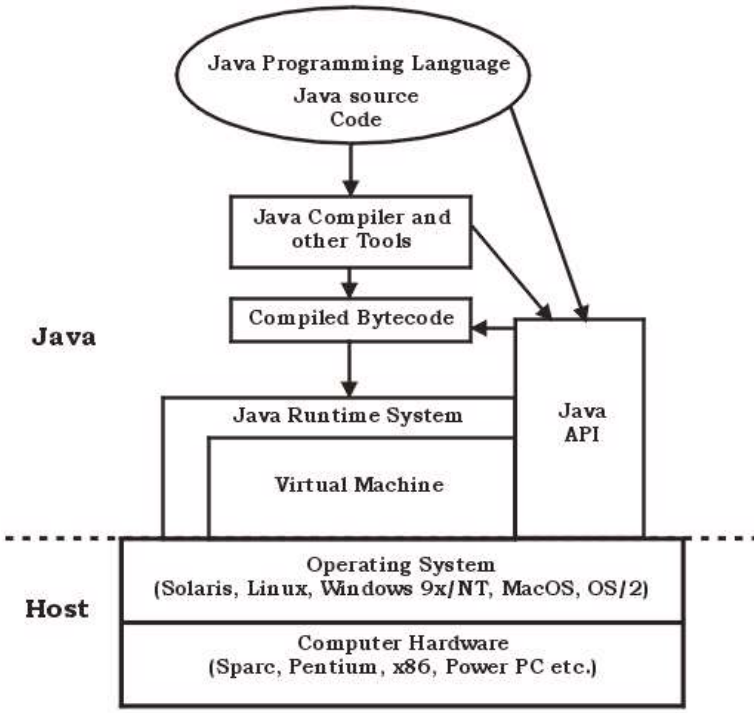


Fig 1.1 : Java unveiled

THE ORIGINS OF JAVA

The origins of Java trace back to 1991, when a research group at Sun was investigating consumer electronics products. They developed a product called Star7 for which they planned to develop an operating system in C++. Then James Gosling, the father of Java, developed a new language for start7, based on C++, eliminating its shortcomings. Initially it was named as Oak but later Sun renamed this language as Java by the year 1994. HotJava, Java, and the Java documentation and source code were made available over the Web, as an alpha version, in early 1995.

In 1996, Java 1.0 was officially released and made available for download over the Internet. JavaScript was also released. Then came Java 1.1 with further enhancements and finally Java 1.2.2 which is also called as Java 2 is the latest version. Netscape 2.0 now provides support for both Java and JavaScript.

JAVA AND HOTJAVA

Java is an object oriented programming language, a runtime system, a set of development tools, and an application programming interface (API) whereas HotJava is a Web browser that is written in Java. Like

HotJava, the Netscape 2.0 browser, is also Java enabled. Netscape, is written in C++ and uses C++ functions to display HTML documents.

When an APPLET tag is encountered in an HTML file, then both Hot Java as well as Netscape requests the Java bytecode file necessary to execute the applet from the Web server. HotJava loads and executes the bytecode file using the Java runtime system whereas Netscape executes the bytecode file using an embedded version of the Java runtime system.

WHY PROGRAM IN JAVA?

Java Is Simple and Easy to learn

Java is a object oriented programming language similar to C++. It is easy to write programs in Java for those who are familiar with C++. Java does not contain the cryptic header files and preprocessor statements of C and C++ thereby making it simple and easy to learn.

Java Is Object-Oriented

Java is totally an object oriented language. In Java, classes and objects are at the center of the language. Everything is written inside the classes. Functions , procedures, structures, unions are not present in Java . Only classes and objects are used in Java thereby making it totally object oriented language. Java supports all the features of object-oriented programming such as class hierarchy, inheritance, encapsulation, and polymorphism.

For writing Java programs, you must build on the classes and methods of the Java API. For developing software in Java, you have two choices:

- Build on the classes you have developed, thereby reusing them.
- Rewrite your software from scratch, copying and tailoring useful parts of existing software.

Java Is Safer, More Reliable and Secure

Java is safer to use than C++ as the usage of pointers is totally eliminated in Java. As a result Java programs are very safe as the possibility of arbitrarily overwriting any memory location or hacking the system resources is not there.

Java programs are reliable because Java does not allow the program coming from the internet and running in the browser, called the applet, to access the local system's file system.Java's reliability extends beyond the language level to the compiler and the runtime system. Compile-time checks both syntactic and semantic errors. Runtime checks are also more extensive and effective thereby offering reliability.

Java is defensive by nature. When an applet arrives at its destination, before it is interpreted, a security check is applied on it thereby ensuring that the Java code, which has arrived, is a true Java code, i.e., it does not contain any viruses. A Java code can also be signed before sending it and the receiver can tally the signatures before receiving it. If the signatures do not match, it means that any other person has hacked the code.

The Java runtime system is designed to enforce a security policy by remembering how objects are stored in memory and enforcing correct and secure access to those objects according to its security rules. It performs bytecode verification by passing compiled classes through a simple theorem prover that either proves that the code is secure or insecure.

Java Is Multithreaded

Java supports multithreading. Multithreading allows more than one thread of execution to take place within a single program. Multithreading enables several different execution threads to run at the same time inside the same program, in parallel, without interfering with each other. Multithreading also allows Java to use idle CPU time, to perform necessary garbage collection and general system maintenance.

Java Is Interpreted and Portable

Java is a truly platform-independent interpreted programming language. Because of its interpreted nature it is portable, although it is slower than its compiled native code. Java programs (.java file) are executed in two stages . They are compiled at the developers end, uploaded on the internet server, then interpreted again and executed. Java programs are compiled into byte code(.class file). Byte code is not specific to any machine, but are written for the JVM. JVM is the software for a virtual machine, i.e., the machine which really does not exist, but in reality is a simulated environment. The byte code is to be embedded inside a web page The web page is uploaded on a specific web site.

The web page is accessed using a browser. If a browser is Java enabled then on accessing the web page Java program begins to execute. On the other hand, if browser is not Java enabled then it would not recognize the applet.

Robust

Java supports techniques such as garbage collection and exception handling to handle situations when a program behaves in an exceptional manner. Memory management issues and exceptional behaviour of the program in different situations, are thus easily dealt by Java, thereby making it a robust programming language. In garbage collection a daemon thread (thread which becomes alive when the system is comparatively free) keeps on freeing the objects

which are not being referenced. Through exception handling any unexpected behaviour, like trying to access an element of an array beyond it's upper bound, which may crash the program, is dealt with exception handling techniques.

Powerful

Java is a very powerful language, which offers you a complete range of classes, with number of small independent functions, so that you can use them in your programs.

JAVA DEVELOPMENT KIT

The Java Developers Kit (JDK) is a comprehensive set of tools, utilities, documentation, and sample code for developing Java programs. The kit lets you work with various tools that are available using command-line option. To check out the latest version of Java, you can log on to www.javasoft.com. This Web site provides all the latest news and information regarding Java, including the latest release of the JDK. You can download the latest release of JDK from the website also. It is downloaded as a compressed self-extracting archive file.

You can simply execute the archive file to automatically create a java directory and build a directory structure to contain the rest of the JDK support files. The JDK support files contains BIN folder, LIB folder for library files, INCLUDE folder and JRE folder.

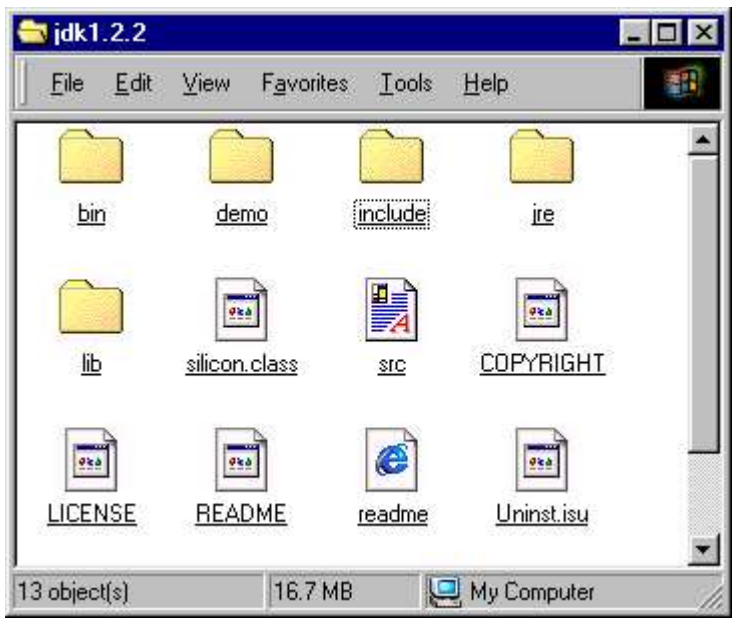


Fig 1.2 The contents of Java Development Kit

The main components in the Java Development Kit lets you develope, test, compile and execute Java programs and applets.

javac	Compiling the Java Code
java	Runtime Interpreter
jdb	Debugging the code
javap	Disassembler
appletviewer	Applet viewer
javadoc	Documentation generator
javah	C header file generator

Besides the above main components, you also have, Archiver, Digital signer, Remote Method Invocation tools, Sample demos and source code and API source code for the classes.

You can write Java programs using any text editor. The source file uses the extension .java and consists of source code packages to declare the source code packages and interfaces.

THE COMPILER

You must compile the source code to execute it. You can use Java compiler, javac, convert the source files into **bytecode files.** These files end with .class extension and can be executed using the Java interpreter.

If you have defined more than one class in the source code, compiler will generate separate bytecode files for each class defined. The Java compiler is a command-line utility, the general syntax of which is as follows:

javac Options Filename

The Filename argument specifies the name of the source code file you want to compile. The Option argument specifies the way , the compiler will create the executable Java class. Java compiler can take the following options:

-classpath Path	To override the CLASSPATH environment variable, you can define the path with -classpath option.
-d Dir	The -d option lets you define the root directory where compiled classes are stored. This is important when classes are organized in a hierarchical directory structure. With the -d option, you can create the directory structure under the directory specified by Dir.
-g	The -g compiler option tells the compiler to generate debugging tables for the Java classes. These tables contains information for Java debugger.
-nowarn	The -nowarn option turns off compiler warnings.
-O	The -O option causes the compiler to optimize the compiled code. The -O optimization option also

suppresses the default creation of line numbers by the compiler.

Setting Classes

Every Java program uses classes that are defined outside of the program's source code file. Most often these external classes are contained in the Java API. However, you will also want to be able to reuse classes that you develop. The Java compiler must be able to locate these external classes in order to determine how to correctly compile code that references them.

To enable Java Compiler to locate the classes easily, you must set the CLASSPATH. **Set CLASSPATH** statement tells the JDK programs about the location of external classes. In Windows based system, you can set the CLASSPATH by executing the following statement at console command line:

set CLASSPATH=path

A common CLASSPATH is **.;c:\jdk1.2.2;c:\jdk1.2.2\lib\classes.zip**. This tells the Java compiler and other JDK tools to use the current directory (**.**), the **c:\jdk1.2.2** directory, and the file **c:\jdk1.2.2\lib\classes.zip** as a base for finding Java class files. You would enter the following to set this CLASSPATH:

set CLASSPATH=.;c:\jdk1.2.2;c:\jdk1.2.2\lib\classes.zip

You can also put this statement in your AUTOEXEC.BAT file so that it will be set automatically each time you start a DOS shell. Windows NT users can use the System option in the Control Panel to set the CLASSPATH variable.

Also set the path as path= .;c:\java1.2.2\bin; c:\java1.2.2\lib; c:\java1.2.2\include, where java1.2.2 is the directory (Default directory for JDK ver 2.0) storing Java software.

THE RUNTIME INTERPRETER

To execute the compiled files - bytecode files with .class extension, you use the Java interpreter, **java**. You may note that runtime interpreter is a command-line tool only for Non-graphical Java Programs, as graphical programs require the display support of a browser.

The Interpreter verifies the integrity, correct operation, and security of each class as it is loaded and executed. It also interacts with the operating system, windowing environment and communication facilities to produce the desired program behavior.

The syntax for using the runtime interpreter follows:

java Options Classname

The Classname specifies the name of the class, you want to execute. The Option argument controls the way, the runtime interpreter

executes the code. Following is the list of most important runtime interpreter options:

-debug	Start the interpreter in debugging mode.
-checksource, -cs	causes the interpreter to compare the modification dates of the source and executable class files.
-classpath Path	lets the interpreter to override CLASSPATH with the path specified by Path.
-verbose, -v	causes the interpreter to print a message to standard output each time a Java class is loaded.
-verify	causes the interpreter to run the bytecode verifier on all code loaded into the runtime environment.
-noverify	turnsoff all code verification.
-DPropertyName=NewValue	enables you to redefine property values.

THE DEBUGGER

This utility debugs the Java applications. Java Debugger is similar to the interpreter as it also executes the bytecode files but it also provides capabilities to stop program execution at selected breakpoints and to display the values of class variables, thus making it capable of finding error in the programs. The syntax for using the Java debugger follows:

jdb Options

After starting **jdb**, it will announce that it is initializing, identify the class that it is debugging, and print a simple prompt (>). Once you are up and running, using the debugger, you can enter a question mark (?) at the prompt (>) to receive a list of available debugging commands.

THE DISASSEMBLER

Here is the tools which gives you the source code from the compiled bytecode files. The disassembler takes the bytecode files and displays the classes, fields (variables), and methods that have been compiled into the bytecodes. This tool plays a major role for recovering the source code design of those compiled Java classes for which no source code is available - for example, those that you would retrieve from the Web. The debugger is executed as follows:

javap options class

The debugger takes the full name of a class as its input and identifies the variables and methods that have been compiled into the class's bytecodes. It also identifies the source bytecode instructions that implement the class methods.

The option control the type of information displayed by the disassembler. When no option is used, only public fields and methods of the class are displayed. Other options are described below:

-p Also displays the class's private and protected fields and methods.

-c displays the source bytecode instructions for the class methods.

-classpath Overrides the classpath settings.

THE APPLET VIEWER

You can use the applet viewer, appletviewer, to display Java applets contained within Web pages located on your local file system, or at accessible websites. You can also use it to test applets that you develop.

The applet viewer creates a window in which the applet can be viewed. It provides complete support for all applet functions, including networking and multimedia capabilities.

You can start the applet viewer from a command line using the following syntax:

appletviewer Options URL

Here URL is the address of the html page which embedded the Java applet. The options argument specifies the various options that you can use while running the applet. The appletviewer supports only one option:

-debug starts the applet viewer in the Java debugger, which enables you to debug the applet.

e.g. appletviewer silicon.html

Here silicon.html is the HTML file containing the embedded Java applet.

THE DOCUMENT GENERATOR

You can use javadoc- the automated documentation tool - to generate API documentation directly from Java source code. The document generator scans through the Java source code and generates HTML source code based on declarations and special comments inserted into these files. The document generator takes the following syntax:

javadoc Options FileName

Here filename could be the name of the package, source code file or class file name. The options argument specifies the various options that you can use while running the javadoc. The document generator can take the following options:

-classpath Overrides the CLASSPATH environment variable.

-d	Use it to specify the directory to which the generated HTML files are to be stored.
-verbose	Results in a comprehensive display of the files that are loaded during the documentation-generation process.

HEADER FILE GENERATION

You use the javah - the header file generator tool - to generate the C-language header and source files from a Java bytecode file. The header file contains C-language definitions whose layout is similar that of the corresponding Java class. The source files contain the stubs for C functions whose layout is similar to class methods.

You can use the following syntax to invoke the header and stub file generator:

javah options classname

Here, classname refers to the full class names of the classes for which header files are to be generated. The options argument specifies the various options that you can use while generating header and stub files. The header and stub file generator can take the following options :

-o	combines all the files generated into a single file.
-d	specifies the directory where javah is to put the generated header and source files.
-td	specifies the directory where javah stores temporary files used in the header file generation process.
-stubs	causes javah to generate C function stubs for class methods.
-verbose	is used to display status information concerning the files generated.
-classpath	is used to overrides the CLASSPATH environment variable.

PROGRAM TYPES IN JAVA

The Java language basically has two types of programs, the applets, and the applications. Applets are small programs that have the capability to run on the internet through a browser, whereas the applications do not possess this capability. Browsers that are Java enabled can only run Java applet. The Internet Explorer 3.1 and above are able to run the applets successfully.

APPLICATIONS

Java is an object-oriented language. This means that the language is based on the concept of an object. Everything is written in the form of classes. The class names is case sensitive. A Java application

means that it runs within the Java interpreter as a stand-alone program and has text output.

```
public class Silicon
{
   public static void main (String args[])
   {
   System.out.println("Silicon Media!");
   }
}
```

Listing 1.1 A Simple Java Application

Java programs can be written in the DOS editor- edit, or notepad. The file should be saved with the same name as the initiating class name with the .java extension, like in the above case it is 'silicon.java'. The public access specifier is given with the class so that the class is visible to the JVM.

The first statement of Silicon tells you that Silicon is a class, not just a program. This name is used by the Java compiler as the name of the executable output class. The Java compiler creates an executable class file for each class defined in a Java source file. If more than one class are defined in a .java file, then Java compiler will store each one in a separate .class file.

The 'main()' is the first method automatically executed by the complier. From this method all the other classes and their methods are linked. The static keyword main() is given so that the JVM is able to access the main() method directly without instantiating the class that contains the main(). The keyword void is given, as the main() does not return anything. The System.out.println() is a method to print on the output stream.

After compiling the program with the Java compiler (javac), you are ready to run it in the Java interpreter. The Java compiler places the executable output in a file called Silicon.class. This file does not have an extension .EXE although it is an executable file.

```
c:> javac Silicon.java
```

After compilation a .class file is made which contains the byte code.

To run the test program, type java test at the command prompt. The program responds by displaying "**Silicon Media!**" on your screen. It is executed as:

```
c:> java Silicon
```

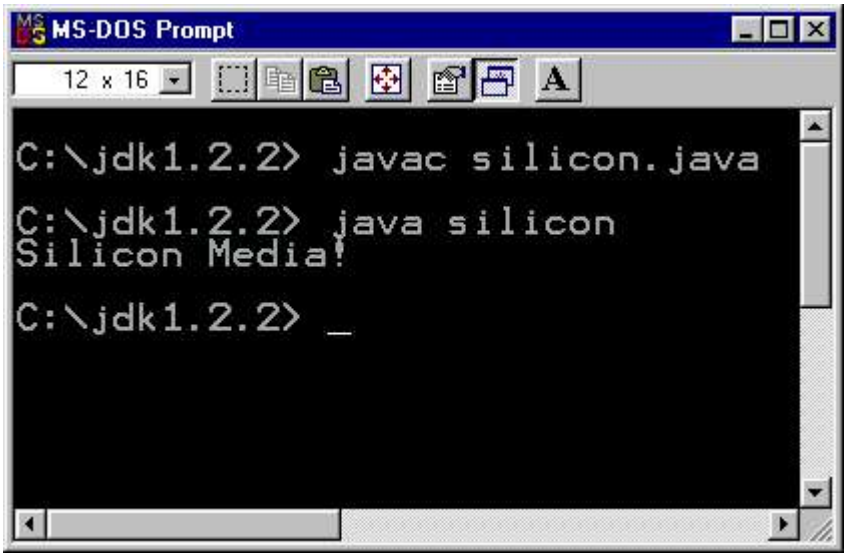


Fig 1.3 Compiling and Running the Java program

APPLETS

Java applets are small programs or mini-applications which are designed to be small, fast, and easily transferable over network resources. Applets generally focus on one small task or a component of a task. Applets are precompiled bytecode. Applets are compiled Java programs that are run using the Applet Viewer or through any World Wide Web browser that supports Java. Applets can display graphics, play sounds, accept user input, manipulate data just like any program, perform interactive programming, such as games, used in animation, displaying Graphic effects, such as scrolling text etc.

Applets consist of distinct stages such as init, start, stop, and destroy. All of these stages can be overwritten by the applet programmer.

A Java applet may contain following four stages:

init();

Initialization of any methods and variables takes place at this stage. The init method is called when the applet is first loaded.

start();

This stage starts the primary functions of an applet. For example, if an applet plays a sound, it could start playing during this stage. This method is called when the init method has finished and the applet is ready to execute.

stop();

This stage is used to stop any actions that is still in progress from the start stage when an applet is exited. For example, a sound loop would need to be stopped when a person left the Web page containing a sound applet. This method is called when the page is left, or the browser is minimized.

destroy();

Destroy is called automatically, and completes any memory cleanup or garbage collection that needs to be done when an applet is destroyed. This method is called when you quit a browser.

An applet consist of following two components :

- The compiled bytecode
- An HTML file containing basic applet information and parameters

The compiled bytecode is the executable component of the applet. In order to execute the code properly the HTML file is required by both the Applet Viewer and any Web browser . The HTML file is based on the <applet> tag and takes the following basic structure:

```
<html>
<applet codebase = "location of code" code= "filename.class" width=100 height=150 alt=alternate>
<param name= "parameter" value= "accepted value">
</applet>
</html>
```

The <applet> tag contains the filename of your executable code, in the format of filename.class, followed by the dimensions of your applet. The initial dimensions of the applet need to be given in pixels. The <param> tag accepts the parameter name and its associated value. The parameter tag is designed to take parameters, and it can be repeated as many times as necessary to establish the parameter values.

Similar to the applications, an applet can be written with a class having the same name as the filename. There are various other methods present in an applet.

HOW APPLETS DIFFER FROM APPLICATIONS

There are significant differences between applets and applications. Applets are not full-featured applications. There are concrete and implicit limitations that need to be considered in designing and creating applets. For example:

- Applets have limited file access : The applications can access the local file system and the resources, whereas applets are not given full access to the local machine's file system.

Applets do not have the means to save files out to a user's local machine, and they can't read files off the local machine either.

- Applications are trustworthy as you know the person who has handed over the program to you and are aware of its functionality. Whereas in the case of applets, you are totally unaware of the fact that who is the author of the program, and whether it is safe enough to run the applet.
- Applets have limited network access

EXERCISE

1. What kind of tool runs a computer program by figuring out one line at a time?
   (a) A slow tool
   (b) An interpreter
   (c) A compiler
2. When you compile a Java program, what are you doing?
   (a) Saving it to disk
   (b) Converting it into a form the computer can better understand
   (c) Adding it to your program collection
3. What special HTML tag is used to put a Java program onto a Web page?
   (a) <APPLET>
   (b) <PROGRAM>
   (c) <RUN>
4. Which type of Java program can be run by the java interpreter tool?
   (a) Applets
   (b) Applications
   (c) none
5. What is Java? Where did Java come from? What is the role of World Wide Web in development of Java?
6. What platforms does Java run on?
7. What's the difference between an application and an applet?
8. Discuss the various stages in developing Java applets.
9. Discuss the various applications available in Java Development Kit.
10. What is the use of setting CLASSPATH option?
11. Why must the name main() only appear once in your program?

JAVA LANGUAGE

IDENTIFIER
KEYWORDS
DATA TYPES
PRINTING DATA
CONSTANTS IN JAVA
BLOCK AND SCOPE
EXPRESSIONS AND OPERATORS
CONTROL STRUCTURES
BRANCHING
THE SWITCH STATEMENT
LOOPING

Java Language

Java is object oriented language, similar to C++ or smalltalk. If you have idea about these languages, you will realize in the coming chapters that it is modified form of C++. The listing 1.1 gives enough idea about the similarity between C++ and Java.

IDENTIFIER

An identifier is a user defined name that identifies some storage.

Identifiers are used to name variables, constants, functions, arrays, structures etc. Java identifiers are case-sensitive and must begin with a letter, an underscore (_), or a dollar sign ($). Both upper- and lowercase characters can be used to create identifier names. Subsequent identifier characters can include the numbers 0 to 9. Two different identifier names can have same spellings, one in upper case and other in lower case, as Java is a case sensitive language. The Java keywords should not be used as an identifier.

KEYWORDS

Keywords are predefined identifiers reserved by Java. The following keywords are reserved for Java:

abstract	double	int	super
boolean	else	interface	switch
break	extend	long	synchronized
byte	false	native	this
byvalue	final	new	threadsafe
case	finally	null	throw
catch	float	package	transient
char	for	private	true
class	goto	protected	try
const	if	public	void
continue	implements	return	while
default	import	short	do
instance	of	static	

DATA TYPES

Data Types forms the basis of any programming language. Programming languages can represent data in a variety of ways. Data types define the storage methods available for representing information.

The syntax of the Java declaration statement for variables follows:

Type Identifier [Identifier];

The declaration statement tells the compiler to set aside memory for a variable of type Type with the name Identifier. The bracketed Identifier indicates that multiple identifiers of the same type can be declared here separated by commas. The square bracket indicates that it is optional. You can make multiple declarations of the same type by separating them with commas. Finally, the declaration statement ends with a semicolon.

In Java, the data types are classified into two categories: simple and composite. Simple data types are core types that are not derived from any other types. Integer, floating-point, Boolean, and character types are all simple types. On the other hand, composite types, are derived from simple types, and include strings, arrays, classes etc.

INTEGER DATA TYPES

An integer is a whole number without having a fractional part. It may have a positive or negative value. In case of either type of machine (16-bit or 32 bit), the left most bit is most significant and is reserved to indicate whether the number is positive or negative.

There are four integer types: byte, short, int, and long depending upon the amount of space required by them in memory.

Data Type	Size	Range
byte	8 bits	-128 to 127
short	16 bits	-32,768 to 32,767
int	32 bits	-2,147,483,648 to +2,147,483,647
long	64 bits	-2^{63} TO 2^{63} -1

The following are some examples of declaring integer variables:

```
int i;
short x;
long seconds;
byte red;
```

FLOATING-POINT DATA TYPES

Floating-point data types are used to represent numbers with fractional parts. There are two floating-point types: float and double. A double is simply a floating point value specified with greater exactness. It usually requires twice as much storage space as a float does. It improves the accuracy of arithmetic and reduces the accumulation of rounding errors. The float type reserves storage for a 32-bit single-precision number whereas the double type reserves storage for a 64-bit double-precision number. Declaration of floating-point variables is similar to declaration of integer variables. The following are some examples of floating-point variable declarations:

```
float avg;
double speed;
```

BOOLEAN DATA TYPE

The boolean data type store values with one of two states: true or false. A Boolean data type is declared in the following way:

```
boolean flag;
```

CHARACTER DATA TYPE

The character data type is used for storing only single characters. You create variables of type char as follows:

```
char choice='y';
```

STRINGS

Strings are handled in Java by a special class called String. This method of handling strings is very different from languages like C and C++, where strings are represented simply as an array of characters. The following are a few strings declared using the Java String class:

```
String account;
String name = "Mr. James";
```

ARRAYS

Computer process data. Quite often the data are organized in some orderly manner - for example, a statistical time series, or a list of name in alphabetic order. A collection of data, whose elements form an ordered sequence, is called array. The elements of an array can be of any data type. Thus array is a secondary data type which can be derived from a primary data type.

Now, the question arises, why use arrays when we have primary data types. Consider an example, where you have to calculate the sum of 100 numbers. Answer is simple - Start a loop, every time give the value of x and add it into the previous sum and after repeating the loop 100 times, we get the required sum.

Here, the problem is that whenever a new value is given for x, the previous value is deleted (after being added into sum) and no longer remains in the memory, thus can't be reused. Suppose after calculating the average, you have to compare the every number with the average of 100 numbers. In such case, you will either input the values again or store all the numbers in separate variable, thus define 100 variables. Both the methods are cumbersome.

In such cases arrays play a vital role and the same array variable can take the defined number of values and store them simultaneously for reuse.

An array provides for the storage of a list of items of the same type. Items in an array can be of either a simple or composite data type. Arrays also can be multidimensional.

One dimensional array

When a list of items is dependent on only one variable name and can be created using only one subscript, such a variable is called a single-subscripted variable or a one-dimensional array.

e.g. $\sum_{i=1}^{n} x_i$ **to calculate the sum of n values of x.**

Here only one variable x with a subscript, takes different values and its sum is calculated. The subscripted x; refers to the ith value of x. The subscript must be non-negative integer. In Java, single subscripted variable xi can be expressed as

x[1], x[2], x[3] ..... x[n]

Thus the same variable takes different values using arrays.

Declaration of array

Like all other variables, arrays is also declared before they are used. The general form for array declaration is :

Data type variable-name [];

Arrays are declared with square brackets ([]). The following are a few examples of arrays in Java:

```
int num[];
char[] name;
long xyz[][];
```

Square brackets can be placed either after the variable type or after the identifier. As it is obvious from above examples, Java doesn't allow you to specify the size of an empty array when declaring the array. This is because Java does not support pointers and thereby there is no way of pointing anywhere in an array. The size of the array can be set explicitly either by using a new operator or by assigning a list of items to the array on creation.

Following examples illustrates that how arrays can be declared and set to a specific size by using the new operator or by assigning a list of items in the array declaration:

```
char letters[] = new char[26];
int odd = {1, 3, 5};
```

Two dimensional array

In case of single dimension array, the variable can take only one subscript. At times, it is required to store a table into variable e.g. data of matrix. In such case variable requires two subscript such as x_{ij}

e.g. $\sum_{i=1,j=1}^{n,m} x_{ij}$

Java allows to define such tables of items by using two-dimensional array. The table or matrix form can be defined in Java as

X [][]

Two dimensional arrays are declared as follows :

type array-name [][]

Two dimensional arrays are stored in the memory as shown below

	Column 0	Column 1	Column 2
Row 0	**[0][0]**	**[0][1]**	**[0][2]**
Row 1	**[1][0]**	**[1][1]**	**[1][2]**
Row 2	**[2][0]**	**[2][1]**	**[2][2]**
Row 3	**[3][0]**	**[3][1]**	**[3][2]**

Initializing Two-dimensional Array

Like one dimensional array, two dimensional array may be initialized by by using the new operator :

Data type array-name[][] = new Data-type[row][column];

Consider the following example, where esal array is being intialized and value being assigned to the variable.

```
String esal[ ][ ] = new String [1][3];
esal[0][0] = "M.N. Gulati";
esal[0][1] = "Ambica Chawla";
esal[0][2] = "Sanjeev Chawla";
```

CASTING TYPES

The process of converting one data type to another data type is called casting. It is often required when a function returns a type different than the type you need to perform an operation. For example, the read member function of the standard input stream (System.in) returns an int. You must cast the returned int type to a char type before storing it, as in the following:

```
char c = (char)System.in.read();
```

In order to perform casting the desired type is placed in parentheses to the left of the value to be converted. The System.in.read function call returns an int value, which then is cast to a char because of the (char) cast. The resulting char value is then stored in the char variable c.

While casting, care should be taken that the destination type should always be equal to or larger in size than the source type so that there is no loss of information.

Casts that result in no loss of information.

From Type	To Type
byte	short, char, int, long, float, double
short	int, long, float, double
char	int, long, float, double
int	long, float, double
long	float, double
float	double

PRINTING DATA

Consider the program listing given below:

```
public class Silicon {
    public static void main (String args[])
    {
    System.out.println("Silicon Media!");
    }
}
```

As discussed earlier, the first statement declare Silicon as a class. Next the 'main()' is the first method automatically executed by the compiler. The opening brace '{' in the third line marks the beginning of function main and the closing brace '}' in the last line indicates the end of the method. All the statement between two braces form the method body. This method body contains a set of instructions to perform the given task.

Here, this method contains only one statement **System.out.println**, which is predefined, standard Java method for printing output. The System.out.println command means "display a line on the system output device". In most cases, the system output device is your computer monitor. Everything within the parentheses is displayed.

To display the numbers using **println()**, you can use + sign to add the variable into the string.

```
int year = 2001;
System.out.println("Silicon Media! " + year);
```

The output of above println() statement will be

Silicon Media! 2001

CONSTANTS IN JAVA

In order to refer to a value which does not change throughout the execution of a program, Java language provides the usage of constants.

There are several categories of constants in Java language. They are classified as-

- Integer constants
- Floating point constants
- Character constants
- String constants

INTEGER CONSTANTS

Java can accept integer constants written in either the decimal, octal or hexadecimal number system.

The decimal constants are written directly as per the value. The octal values must be preceded by a zero & The Hex values must always be preceded by a zero and either a small X or uppercase X.

e.g.

```
int a;
int b;
int c;
a = 10;
b = 0737;
c =0X7FFF;
```

The above method of initializing a variable is correct but lengthy, where it is known that variable is to be started with a certain initial value, the above two operations can be combined as shown below :

```
int a = 10;
int b = 0737;
int C = 0X7FFF;
```

Floating Point Constants

As discussed earlier the floating point constants can be defined using the exponential notation. The general format for this notation is inc part.fracpart intexp.

e.g. float rank = 4.23e+03

In above example 4.023 is **incpart**, e is **fracpart** and +03 is **intexp**.

Although each of the component is the notation for a floating point constant is optional, and absence of certain components dictates the presence of others and vice/versa. Either the int part or fracpart (but not both) may be missing or either the decimal point or e and intexp (but not both) may be missing.

Character Constant

A character constant can be any of the valid printing character is ASCII character set enclosed in single quotes (‘ ‘). They are assigned through an assignment statement just as in the case of integer and floating point constants.

e.g. char grade = ‘A’

Thus character constants can be written in above form. But the special character like tab or back space or enter can't be written in above cited way. For most common special character, C provides what are called escape sequence - the back slash character \ followed by a letter. The backslash \ signals the compiler that the letter which follows it has special significance. The escape sequence defined for C are :

\n	**new line (i.e. linefeed - carriage return)**
\t	**tab**
\b	**backspace**
\r	**carriage return**
\f	**form feed**
****	**backslash**
\'	**single quote**
\"	**double quote**
\f	**form feed**
\udddd	**unicode character**
\ddd	**octalc character**
****	**continuation**

The above is called **escape sequence** and the different escape sequences can be mixed freely with other character in coding.

e.g. **System.out.println ("A\n BC\n DEF\n")**

will generate the output

A
BC
DEF

when the above statement is executed.

String Constants

A string constant is a list of characters, enclosed in double quotes (" "). Like constants discussed earlier, a string constant is also assigned to an identifier through an assignment statement. The compiler itself adds a null byte (i.e. \0) to the end of a string. This null-byte is used by string handling functions to determine the end of the string.

e.g. **"This is a string constant"**

In case the double quote is to be incorporated in the string it must be preceded by a backslash.

COMMENTS

Comments basically can be written down in two ways. These are the '//' and '/* */'.

// is used to comment a single line. It is similar to the comments used in C++. It can be placed anywhere in the line and the compiler would ignore anything following it. /* */ is borrowed from C and is used to mark a block as a comment entry. This is very useful when the comment entry extends several lines.

There is another comment type (/** comment **/) which works in the same fashion as the C-style comment type, with the additional benefit that it can be used with the Java documentation generator tool, javadoc, to create automatic documentation from the source code.

BLOCK AND SCOPE

In Java, source code is divided into parts using opening and closing curly braces: { and }. Anything between curly braces is considered as a block which is independent of everything outside the braces. Blocks are integral part of Java language syntax. Blocks in Java can be nested, which means that code can be divided into individual blocks nested under other blocks. One block can contain one or more nested subblocks.

The concept of scope is linked with the life of variable in Java. The scope of a variable is defined by the block in which it is declared. The variable can be used from the block in which it is declared, to all subsequent blocks inside it. But the same variable outside the block can not be used. Consider the following set of instructions:

```
public class Varscope
        {
        public static void main(String args[])
        {
            {
            int block1 = 100;
                    {
                    int block2 = 200;
                    System.out.println("Block 1 variable " + block1);
                    System.out.println("Block 2 variable " + block2);
                    }
            block1 = block1+ 1;
            System.out.println("New block 1variable " + block1);
            block2= block2 + 1;
            }
        }
        }
```

In the above listing, two variable block1and block2 are defined in the outer and inner blocks respectively. The variable block1 can be used in both the blocks as it has been declared in the outer block. But the variable block2 can be used only in the inner block, as it has been

declare in inner block. As soon as, you try to access block1 outside the inner block you receive an error message.

EXPRESSIONS AND OPERATORS

In addition to variable and types, all programming languages support the concept of operators. Operators enable you to perform an evaluation or computation on a data object or objects. Right now the concentration will be on following operators.

- Arithmetic Operators
- Modulus Operators
- Increment and Decrement Operators
- Relational Operator
- Logical Operator
- Bitwise Operator
- Boolean Operators
- Assignment Operator
- String Operator

Before, we discuss about operators, let us discuss about expression.

An expression consists of references to previously defined values such as identifiers and constants. Expression usually contain one or more operators. These operators and expressions are evaluated, according to the standard Java language rules. An operator may take one or more operands and operands in turn can consist of previously declared identifiers, constants, functions etc.

An example of an expression follows:

```
x = y * 5;
```

In this expression, x and y are variables, 5 is a literal, and = and * are operators. This expression states that the y variable is multiplied by 5 using the multiplication operator (*), and the result is stored in x using the assignment operator (=).

ARITHMETIC OPERATORS

The usual integer operations found in most of the programming languages are also available in Java. These include addition, subtraction, multiplication and division.

Following table shows the Java language arithmetic operators and their descriptions:

Operation	Symbol	Notes
Addition	+	
Subtraction	-	
Multiplication	*	

Division	**/**	**remainder lost**
Modulo	**%**	**remainder from division**
Increment	**++**	**placement critical**
Decrement	**- -**	**placement critical.**

Consider the expression

a = 1+ 2;

The statement has two operators, Arithmetic operators + and assignment operator =, and three operands a, 1 and 2. The effect of this statement is to add the constants 1 and 2 together and assign the result to the variable a, which must have declared as int in the beginning of program.

Thus, the assignment operator = in Java is not used like the equal sign of algebra.

The Modulus Operator

Normally, when one integer is divided by another, and there is a remainder left, the remainder is simply thrown away. In Java, 5/3 yields 1 and 3/5 yields 0. If remainder is to be calculated, the modulus operator % must be used, which has the same precedence as the multiplication and division operators.

The modulus operator is a necessary supplement to the division operator. It is used in the same way as / but instead of quotient, the remainder is obtained in the result.

It must also be noted that modules operator works only with int and char values but not with float or double.

INCREMENT & DECREMENT OPERATOR

Increasing and decreasing by 1 is such a common computing practice that Java, similar to C and C++, offers shorthand version of above type of assignment. The Operator for these are ++ & - -. There are two possible placement of these operators. Either operator may be placed immediately before or after a variable name.

```
e.g.    a = b ++;     /* Set a to b, then increase  b by 1*/
        a = b - -;    /* Set a to b,  then decrease  b by 1 */
        a = ++ b;     /* Set b to b+1,  then set a to new b */
        a = - - b ;   /* Set b to b+1,  then set a to  new b */
```

The double symbol ++ and - - after variable are called the post increment and post decrement operators, implying that b is increased or decreased by 1 after the assignment to a. The general term post fix is used for such operators.

Similarly the prefix operators ++ and- - appearing before b are known specifically as pre increment and pre decrement operators. With these, the increment or decrement by 1 is performed on b before the assignment to variable a is made.

To illustrate the above, consider the following example:

```
class IncDec
{
        public static void main (String args[])
        {
        int a = 5;
        int b = 3;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("++a = " +  ++a);
        System.out.println("++b = " +  ++b);
        System.out.println("b++ = " + b++);
        System.out.println("a-- = " + a--);
        System.out.println("++a = " +  ++a);
        System.out.println("++b = " +  ++b);
        }
}
```

The output of the above program comes as follows:

```
a = 5
b = 3
++a = 6
++b = 4
b++ = 4
a-- = 6
++a = 6
++b = 6
```

The postfix and prefix methods of increment and decrement are effective enough without using any assignment.

```
b++;                    /*Set b to b+1*/
++b;                    /*Set b to b+1*/
b - -;                  /*Set b to b-1*/
- - b;                  /*Set b to b-1*/
```

As shown in the table of precedence and associativity the ++ & - -, both operators have R-L (Right to Left) associativity. Thus b++=a is not a valid statement of Java.

RELATIONAL OPERATORS

You can use relational operators, whenever you want to compare two quantities or variables and on the basis of that, you want to take some decision. The expression containing the relational operator is called

relational expression and such expression can have a value of one (for true relation) or zero (for false relation).

e.g. 10 < 20 is true
& 20 < 10 is false

Java has six relational operator. These operators and their meaning are given below -

Operator	Meaning
<	is less than
< =	is less than or equal to
>	is greater than
> =	is greater than or equal to
= =	is equal to
! =	is not equal to

A simple relational expression contains only one relational operator and its general syntax is:

arth.exp1 relational operator arth.exp2

The arith.exp on both sides of the expression may be simple constant, variables or combination of them. These arithmetic expressions are first evaluated before comparing the two sides, as arithmetic operators have a high priority over relational operators (Refer to table of precedence).

```
class Compare
{
        public static void main (String args[])
        {
        int a= 5, b = 3, c = 7;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("a > b = " + (a > b));
        System.out.println("a > c = " + (a > c));
        System.out.println("c > b = " + (c > b));
        }
}
```

The output of the above class is as follows:

```
a = 5
b = 3
c = 7
a > b = true
a > c = false
c > b = true
```

You can note that println() method prints the boolean result after using the relational operators.

LOGICAL OPERATORS

Logical operators are used to test more than one relational expression and lets you make decision on the basis of that. You can use following logical operators in Java:

&&	**Logical AND**
||	**Logical OR**
!	**Logical NOT**

e.g. **price < 40 && profit > 10**

which says that price must be less than 40 and profit must be more than 10, if both the condition are to be met. Such expressions, which combines two or more relational expressions, are termed as a logical expression or a compound relational expression.

These logical expression also yields a value of one (for true logical statement) or zero (for false logical statement), according to the truth table shown below.

Truth Table

Operator-1	Operator-2	&&	||
1	1	1	1
1	0	0	1
0	1	0	1
0	0	0	0

In above example, the logical expression given true only if price is less than 40 and profit is more than 10. If any of the two condition yields a false value (0), the combined condition will yield a false (0) value.

BITWISE OPERATORS

Java has a powerful set of special operators capable of manipulating data at bit level, called bitwise operators. These operators are used for testing the bits, and can operate on ints and chars but not on floats or doubles. The following are the bitwise operators available in Java.

Operator	Meaning
&	**bitwise AND**
!	**bitwise OR**
^	**bitwise exclusive OR**
<<	**Shift left**
>>	**Shift right**
~	**Bitwise Compliment**

The Bitwise relational operator i.e. AND, OR and XOR compare the each bit binary equivalent of the two numbers. In AND operator if the two bits being compared are 1, the resultant bit will be 1, otherwise 0. In bitwise OR, the resultant bit is 1 if any of the compared bit is 1. In XOR, the resultant will be 1 if exactly one of the bits is 1.

The last three bitwise operators work on the integers and are unary operators. The right-shift operator >>, shift the binary number to the right bit specified integer amount, whereas << shifts the binary number to the left by specified integer amount. The Compliment operator, flips all 0 to 1 and all 1's to 0 thus giving you the 2's complement of the number.

Consider the following example.

```
class Bitwise {
        public static void main (String args[])
        {
        int a = 5, b = 3;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("a & b = " + (a & b));
        System.out.println("a | b = " + (a | b));
        System.out.println("a ^ b = " + (a ^ b));
        System.out.println("a>>  = " + (a >> 2));
        System.out.println("b << = " + (b << 1));
        System.out.println("~a = " + ( ~a));
        }
}
```

The output of the above class will be as follows

a = 5	(Binary 0101)
b = 3	(Binary 0011)
a & b = 1	(compare the above two binary numbers with AND operator you get 0001)
a | b = 7	(With OR operator you get 0111)
a & b = 6	(With XOR operator you get 0110)
a >> = 0	(0101 when shifted to right by two bits, give 0001 = 1)
b<< = 6	(0011 when shifted to left by one bit, gives 0110 = 6)
~a = -6	(gives -0110, the 2's compliment equivalent to -6)

BOOLEAN OPERATOR

Boolean operators act on Boolean types and return a Boolean result. The following type of boolean operators are used.

Operator	Meaning

&	**Evaluation AND**
|	**Evaluation OR**
^	**Evaluation XOR**
&&	**Logical AND**
||	**Logical OR**
!	**Negation**
==	**Equal-to**
!=	**Not-equal-to**
?:	**Conditional**

The evaluation operators (&, |, and ^) evaluate both sides of an expression before determining the result. The logical operators (&& and ||) avoid the right-side evaluation of the expression if it is not needed.

The boolean negation operator changes the value of a Boolean from false to true or from true to false, depending on the original value. The equal-to boolean operator simply determines whether two boolean values are equal (both true or both false). Similarly, the not-equal-to boolean operator determines whether two boolean operands are unequal.

The conditional Boolean operator (?:) is also known as the ternary operator because it takes three items, a condition and two expressions. The syntax for the conditional operator follows:

Condition ? Expression1 : Expression2

In above syntax, the condition is evaluated - if it returns true value, Expression1 is executed else Expression2 is executed.

ASSIGNMENT OPERATORS

In addition to standard assignment operators, Java provides a number of assignment operators that are combinations of assignment operators with various other operators. These are called compound assignment operators. These are a form of shorthand that provides a more concise way to modify a variable.

Consider the following statement, which is common to most programming languages.

Total = Total+Sum;

In Java, using the compound assignment operator, the statement can be coded as follows :

Total + = Sum;

Here + = use two operators to form a compound assignment. The general form of compound assignment operator can be described as follows -

Left value operator = Right-expression;

Where operator can be any one of the ten Java compoundable operators shown below-

Arithmetical	**+(add) - (subtract), x (multiply), /(divide), % (integer remainder)**
Shifts	**<< (left shift), >> (right shift)**
Bitwise	**&(AND). ! (OR), ^ (XOR)**

The general form described above translates into

Left value = left value Operator Right-expression;

e.g.
```
Sum = Sum * factor ;          /* multiply */
Sum *= factor ;
a = a%b;                      /* (Integer remainder) */
a %= b;
input = input + output;
Input  += output;
```

STRING OPERATORS

Similar to boolean type variables, strings can also be manipulated using operators. For joining the strings, you have the concatenation operator (+). The concatenation operator connects the two strings to give a combined string.

Consider the following example:

```
class AddString
{
        public static void main (String args[])
        {
        String ones = "Silicon " + "Media ";
        String twos = "Computer " + "Books ";
        System.out.println(ones+ twos);
        }
}
```

The output of the above program will be:

Silicon Media Computer Books

OPERATOR PRECEDENCE

Every Java operator has a precedence associated with it. Precedence refers to the order in which Java evaluates its operators. Following is a list of all the Java operators from highest to lowest precedence:

```
.       []      ()
++      --      !       ~
*       /       %
+       -
<<      >>      >>>
```

< > <= >=
== !=
&
^
&&
||
?:
=

In the above list, all of the operators belonging to a particular row have equal precedence. The precedence level of each row decreases from top to bottom. This means that the [] operator has a higher precedence than the * operator, but the same precedence as the () operator.

Expressions are evaluated right to left. Consider a following expression

x = (2 * 10) + 12 / 4

The division operation 12 / 4 is carried out first, which leaves a result of 3. Then multiplication operation (2*10) takes place as brackets have higher precedence than +. Then addition is performed 20 + 3 which gives us the result 23. Thus the value of x =23.

CONTROL STRUCTURES

The control structures are used to control the flow of programs. The flow of program is dictated by two different types of constructs: branches and loops. Branches lets you decided on selectively executing a different set of instruction for different conditions. Loops, on the other hand, lets you repeat certain parts of a program for specified number of times. Together, branches and loops provide you with a powerful means to control the logic and execution of your code.

BRANCHING

A language is incomplete if it doesn't have decision making capabilities. These decision making capabilities can be used to execute different set of instructions under different conditions. e.g. a bank may decide on the interest to be given to customer on the basis of the amount deposited in the bank. If amount exceeds a particular value, say x, the interest paid will be 15% or else it will be 10%.

Thus, here a decision is to be made - whether the amount is greater than x or not. If amount is greater than x, calculate the interest with 15% rate of interest or else calculate the interest with 10% rate of interest.

Java has two major decision making tools, which helps in taking decision and instructing computer to execute the right set of instructions. These includes:

- **if....else** statement
- **switch** statement

IF STATEMENT

The **if** statement may be implemented in different forms depending on the complexity of conditions to be tested.

- **if.....else** statement.
- Nested **if..... else** statement.
- **else if** ladder.

SIMPLE IF STATEMENT

The general form of a simple **If** statement is

If test condition
    statement group;
Next Statement;

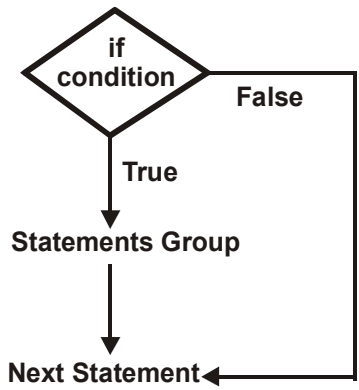


Fig 2.1 If then statement

The statement group could have a single statement or a group of statements, but each statement must be followed by semicolon(;). The condition is defined as entity and it can have two values - true or false. If the test condition is true, the statement group is executed, otherwise, the statement group is skipped and the execution will jump to next statement. If the condition is true both statement group is executed followed by the Next Statement. The following figure represent flow chart for the the If statement :

e.g. Consider the following segment of a procedure

```
.....
if deposit > 4000
        Interest := 0.15* deposit;
```

The above part of the procedure test whether the deposited amount is greater than 4000 or not. If yes, the interest @ of 15% is calculated.

THE IF..... ELSE STATEMENT

The **if.....else** statement is an extension of simple if statement. The general form is

```
if (test condition)
        {
                True statements;
        }
        else
        {
                False statements;
        }
next statement;
```

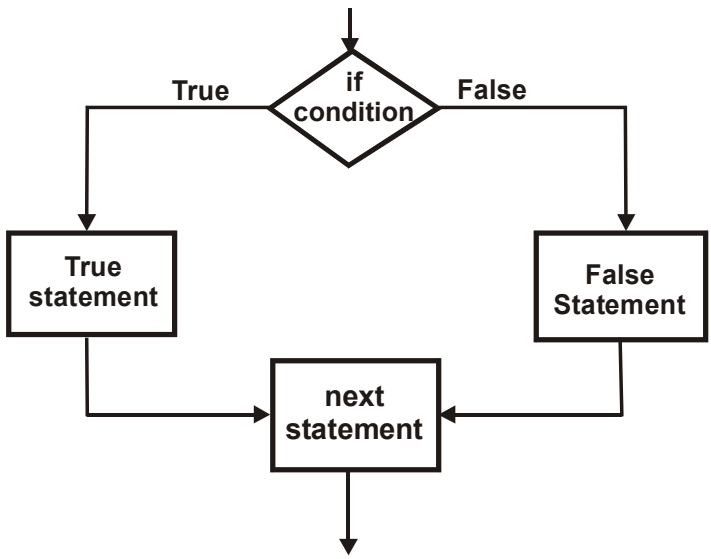


Fig 2.2 If.. Else condition flow chart

Here the difference is that another set of instructions has been added for false condition using else. If the test condition is true, the true statements are executed, otherwise, the control moves to false statements. Thus here one of the two group of statements are executed. In no case, both the sets of instructions can be executed simultaneously. This flow chart is illustrated in the Fig 2.2:

As you can see from the flow chart, only one set of instruction are being executed on the basis of true or false condition. In both the cases, the control is transferred subsequently to **next statement**.

e.g.

```
.....................
```

```
if (code ==”M”)
Mheight = Mheight + height;
else
Fheight = Fheight + height;
System.out.println(“Height: ”);
```

In above program segment, if the code is equal to “M”, the *height* is added to *Mheight* otherwise *height* is added to *Fheight*. In both cases, the control is transferred to prinln statement.

Here you could note down that both the if and else statements are followed by semi colon(;).

NESTED IF.....ELSE STATEMENT

The If statement is termed Nested if statement, if the true statements or the false statements or both contains other if statements. This will happen, when a series of decisions are involved:

```
if (test condition-1)
        {
        if (test condition-2)
                {
                statement group-1;
                }
        else
                {
                statement group-2;
                }
        }
else
        {
        if (test condition-3)
                {
                statement group-3;
                }
        else
                {
                statement group-4;
                }
        }
next statement;
```

The nesting of conditions may be sometimes confusing, it is recommended that nesting conditions always be balanced by having each if clause paired with else, as shown above.

Now let us consider the results produced by the above nested If statement under different conditions -

- If condition 1 is true, the control is transferred to the If-condition 2 or else the control is transferred to If-condition 3.
  - Statement group-1 is executed when both the condition-1 and condition-2 are true.
  - Statement group-2 is executed when condition-1 is true and condition-2 is false.
  - Statement group-3 is executed when condition-1 is false and condition-3 is true.
  - Statement group-4 is executed when both the condition-1 and condition-3 are false.

The flow chart for Nested If is illustrated in the Fig 2.3.

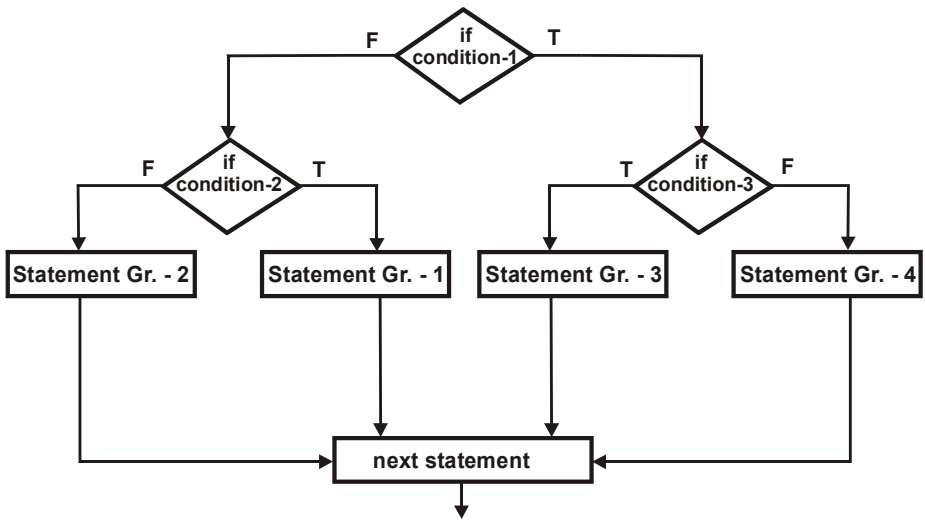


Fig 2.3 Nested If...else flow chart

e.g. To The decides the greatest of three numbers- following set of instructions can be used.

```
.............
    if (A>B)
            {
            if (A>C)
            System.out.println ("Greatest Number " + A);
            else
            System.out.println ("Greatest Number " + C);
            }
    else
            {
            if (C>B)
```

```
                System.out.println ("Greatest Number " + C);
                else
                System.out.println ("Greatest Number " + B);
                }
        .....
```

Other form of NESTED IF..... ELSE Statement

You can also put multiple **ifs** together in a different way. This involves a chain of **ifs** in which the statement associated with each **else** is an **if** statement. Thus, the nesting is taking place inside else portion only. It has the following general form :

```
If (condition-A)
        statement-1;
else    if (condition-B)
                statement-2;
        else    if (condition-C)
                        statement-3;
                else    if (condition-D)
                                statement-4;
                        else
                                statement-x;
next statement;
```

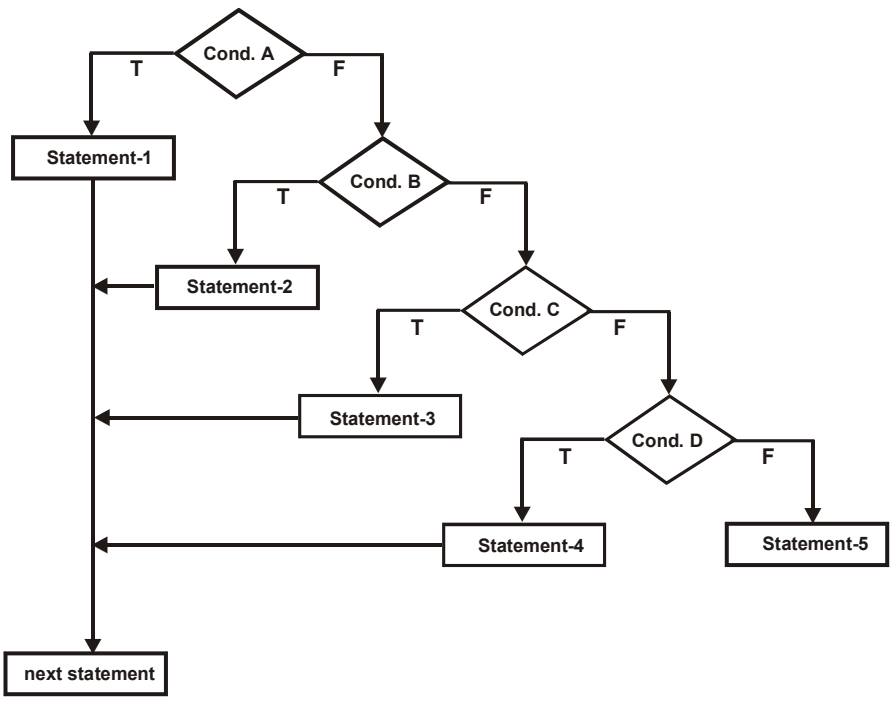


Fig 2.4 If ladder flow chart

This construct is also known as the **else if** ladder. Here, the conditions are evaluated from the top, downwards(A, then B, then C, then D). This type of ladder is used when among many conditions, only one condition is true. As soon as the program encounters the true condition, the statement associated with it is executed and the control is transferred to next statement. If among the defined conditions, no condition is true, then the final **else** containing the *statement-x* is executed.

The flowchart of above conditions can be displayed as shown in Fig 2.4.

The results produced by the above nested If statement under different conditions will be as follows:

True Condition	**Statement Executed**
Condition - A	Statement-1
Condition - B	Statement-2
Condition - C	Statement-3
Condition - D	Statement-4
None	Statement - x

e.g.To decide the student grade on the basis of marks obtained:

Av.Marks	Grade
80-100	**A**
70-79	**B**
60-69	**C**
40-59	**D**
0-39	**E**

You can do the grading using the else if ladder as follows :

```
public class Nest
        {
        public static void main(String args[])
        {
         int marks = 89;
            if (marks>79)
                    System.out.println (" The grade is  " + "A");
            else    if (marks>69)
                    System.out.println (" The grade is  " + "B");
            else    if (marks>59)
                    System.out.println (" The grade is  " + "C");
            else    if (marks>39)
                    System.out.println (" The grade is  " + "D");
            else
                    System.out.println (" The grade is  " + "E");
        }
        }
```

THE SWITCH STATEMENT

Your program may need to make more than two or three decisions based on a single response. A series of If-else statements could do the job, but, then there is an easier way: use the **switch** statement. With the switch statement, the **if-else** tests are made into alternative switches of one long conditional statement. Now, Java chooses the first case, the second case, or whichever other case matches the user's response and executes the block of statement associated with that case.

The general form of the **switch** statement is as shown -

```
Switch (expression)
{
        case value-1;
                statement group 1;
                break;
        case value-2;
                statement group-2;
                break;
        ..........
        ..........
        default;
                default statement;
        break;
}
next statement;
```

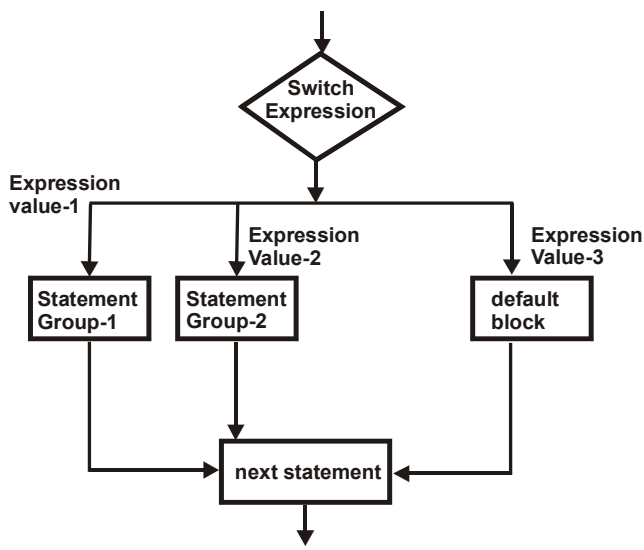


Fig 2.5 Flow chart for switch statement

Here each case is labeled by one or more integer-valued constant or constant expressions. If a case matches the value, the statement group associated with the case is executed.

At the bottom, the default switch is given which is optional. If value of expression doesn't matches with any of the case value, the default statement is executed, if it is present, otherwise no action takes place and control goes to next-statement.

The flowchart of switch statement can be shown as shown in Fig 2.5. Let us consider an example where on the basis of marks obtained in a Combined Entrance Examination, the students will get admission in various Engineering Colleges.

Av.Marks	College
90-100	**IIT-DELHI**
80-89	**IIT-MUMBAI**
70-79	**UNIVERSITY OF ROORKEE**
60-69	**IIT-KANPUR**
50-59	**IIT-NOIDA**
0-49	**TRY YOUR LUCK NEXT TIME**

Consider the following Swtif.java class.

```
public class Swtif
    {
    public static void main(String args[])
    {
     int marks = 89;
    int c = marks/10;
    switch(c)
    {
    case 9:
        System.out.println("Admitted into " + "IIT-DELHI");
        break;
    case 8:
        System.out.println("Admitted into " +"IIT-MUMBAI");
        break;
    case 7:
        System.out.println("Admitted into " +"UNIVERSITY OF
        ROORKEE");
        break;
    case 6:
        System.out.println("Admitted into " + "IIT-KANPUR");
        break;
    case 5:
        System.out.println("Admitted into " +"IIT-NOIDA");
```

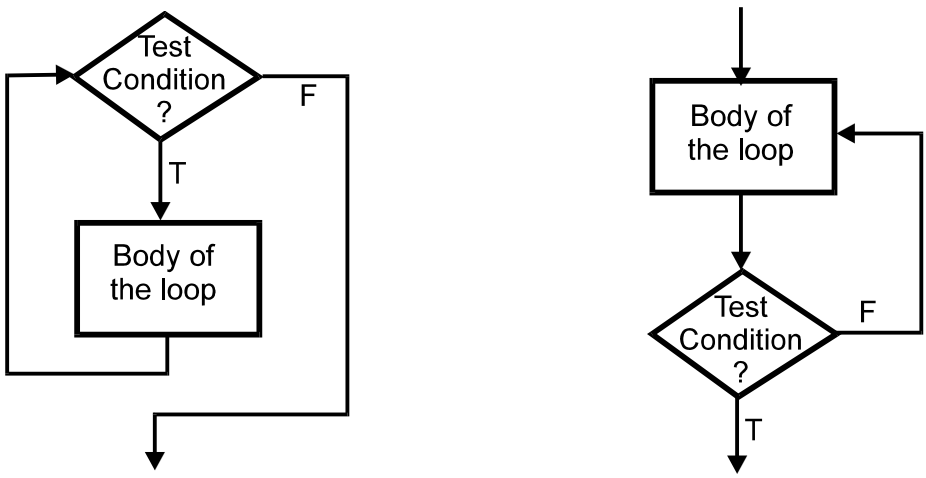
```
            break;
        default:
            System.out.println("Admitted into " +"TRY LUCK NEXT
            TIME");
            break;
        }
        }
}
```

LOOPING

Looping consists of series of statements that are executed until some conditions for termination of loop are satisfied. A program loop consist of normally two segments.

- body of the loop
- Control statement.

The control statement test the conditions and then directs the repeated execution of the series of statements contained in the body of the loop. Thus, a loop is defined as a block of processing steps repeated a certain number of times. An endless loop repeats infinitely and is always the result of an error.



Entry Controlled Loop **Exit Controlled Loop**

Fig 2.6 Flow chart for various loops

Looping can be classified into two categories - Entry-controlled loop or Exit-controlled loop. Entry controlled loop first testify the condition and then executes the series of statements whereas Exit controlled loop first executes the statements and then testify the condition. Or in other words, Entry controlled loop starts with the testing condition and Exit control loop ends with the testing condition.

The general method of looping is by using counters and then testing it using the if statement. Every time the series of statement are executed, the counter increases or decreases its value by a fixed number. And the value of this counter is tested after every execution of series of statements. If the testified condition is not met, the series of statements are executed again and this process continues till the condition is met.

The Java language provides for three loop constructs for performing loop operations. They are

- The **while** statement
- The **for** statement.
- The **do.. while** statement

THE WHILE STATEMENT

This is one of the most basic looping statement. The basic format of while in Java language is:

```
while (test condition)
        {
        body of the loop
        }
```

It is an **entry-controlled loop** statement i.e. test condition is tested before executing series of statements. After execution of the body, the control is transferred back to once again evaluate the test condition and if it is true, the body of the loop is executed once again.

Thus the body of the loop is executed till the test condition with while statement becomes false. When it become false, the control is transferred to the statement immediately after the body of the loop.

The body of the loop may contain one or more statement, but it must include a statement which increases or decreases the value of the variable used in the test condition, thus providing a value which will stop the loop. If no such statement has been used, the loop will continually repeat itself for infinite number of times.

```
public class GN
{
    public static void main(String args[])
    {
    int n = 1;
    int sum = 0;
    while (n <= 10)
    {
            sum = sum + n;
            n++;
    }
```

```
        System.out.println("This Sum is  "+ sum );
        }
    }
```

In above case body of the loop is executed ten times, each time adding the value n to the sum, thus giving the sum of first 10 numbers.

THE FOR STATEMENT

The **for** loop is the most widely used looping statement in Java. It is an entry-controlled loop and is in a very compact and easy form. The general form of **for** loop is :

```
for (initialization; test condition; increment)
        {
        body of the loop
        }
```

Here initialization of the counter (first step in looping), test condition and the increment or decrement statement, all are defined in one line and the body of the loop is defined below the for statement. This makes it most compact looping statement.

In above for statement :

- Initialization — initialize the counter or control variable.
- testing condition — define the test condition that determines when the loop will exit. If the condition is true, the body of the loop is executed, otherwise, the loop is terminated.
- increment — give the expression which modifies, increase or decrease the value of the control variable.

The equivalent **while** statement is given below:

```
initialization;
while(test condition)
{
    body of the loop;
    increment;
}
```

e.g.

```
for (n=1;n<=10; n++)
{
sum = sum + n;
}
```

The above program repeats itself 10 times and each time adds the value n to the sum, thus giving the sum of first 10 numbers.

Nesting of for Loops

Java language allows the nesting of loops, i.e. one **for** statement within another **for** statement as shown below :

```
for (a=1; a<5; a++)
        {       ....
                for (b=1; b<10; b++)
                {       ....
                }
                ....
        }
```

Using for loop in Arrays

Consider the following example which shows the creation and the handling of a two dimension array using a class called EmpSal. This class stores the employee's name and salary in the array and then displays them.

```
public class EmpSal
{
    public static void main(String args[])
    {
    String esal[ ][ ] = new String [2][4];
    esal[0][0] = "M.N. Gulati";
    esal[0][1] = "Ambica Chawla";
    esal[0][2] = "Sanjeev Chawla";
    esal[0][3] = "Preeti Jain";
    esal[1][0] = "9000";
    esal[1][1] = "15000";
    esal[1][2] = "11000";
    esal[1][3] = "8000";
    for(int i=0; i < 4; i++)
    {
            System.out.println("Employee Name " +
            esal[0][i]);
            System.out.println("Salary " +
            esal[1][i]);
    }
    }
}
```

The output of the above Program is shown in Fig 2.7.

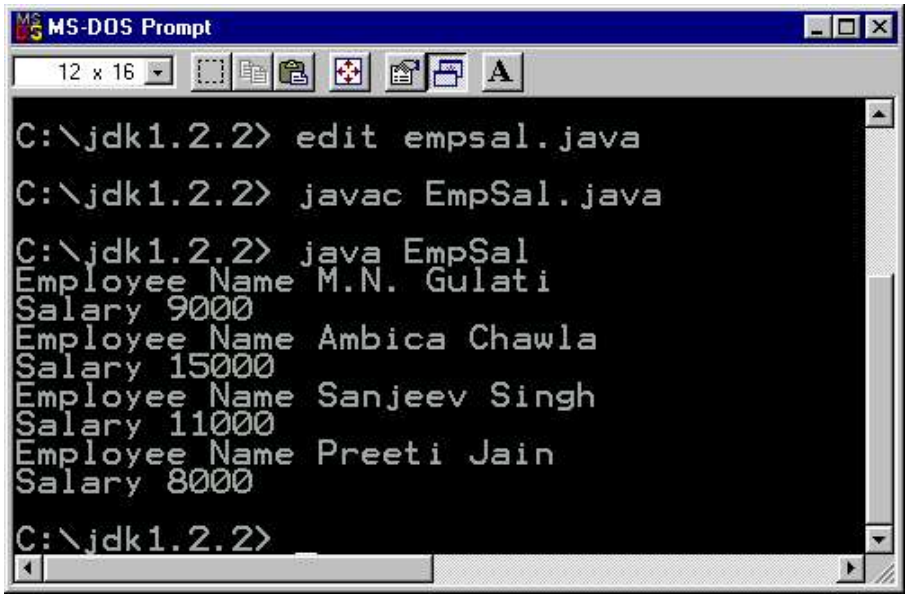


Fig 2.7 Executing Arrays Program

THE DO...WHILE STATEMENT

The **while** loop construct performs the test condition before the loop is executed. If the condition is not met, the set of statements in the body of the loop may not at all be executed. At times, it is required to execute the body of the loop before test is performed. You can handle such situations with the help of do....while statement.

```
do
        {
        body of the loop
        }
while (test-condition);
```

Here, the body of the loop is executed first and at the end of the body execution, the test condition is evaluated. If it returns true value, the program executes the body of the loop again. If the condition returns the false value, the loop is terminated and the control is transferred to the statement that appears just after the while statement.

As the test condition is being executed after the body of the loop, the do.....while statement creates an exit-control loop.

e.g.

```
n =1;
do
    {
    sum = sum + n;
    n++;
    } while (n < 10);
```

Here you could note that while expression takes value n< 10 and not n <= 100. This is because the body of the loop is being executed first and when test condition is being tested, only 9 more loops are required to satisfy the condition. Therefore, test condition tells to repeat the loop as long as the n remains less than 10.

BREAKING LOOPS

Loops performs the set of predefined operations repeatedly until the control variable fails to satisfied. As the number of times, the loop is executed is decided in advance, you can't stop the looping process in middle in normal course. But sometimes it is required to skip a part of Loop or complete loop. e.g. consider the case of searching for a particular name in a list containing 10 names. A program loop written to read and test the 10 names, but it must be terminated as soon as the desired name is found.

Using Java language, you can

- Jump out of loops
- Skip a part of Loop

Jumping out of Loop

You can use the **break** statement to exit the loop after a particular condition is satisfied. As we have discussed earlier in case statement, the break statement breaks the process being executed and transfers the control to the next statement following the case statement.

Similarly, here also, the **break** statement breaks the loop and transfers the control to the statement immediately following the loop. In case of nested looping, **break** statement would only exit from the loop containing it i.e. the break will exit only a single loop.

The general form of break and continue statement can like shown below

```
for (.............)
{
.................
if (test condition)
    break;
}
......
```

Skipping a part of Loop

During the loop execution, sometimes it is necessary to skip a part of Loop. Such condition may arise when for a particular condition, the execution of Loop is not required.

Java language provides continue statement for this purpose. The continue statement causes the loop to be continued with the next

iteration after skipping any statements in between. The continue statement tells the compiler to :

> "SKIP THE FOLLOWING STATEMENTS IN THE BODY OF LOOP AND CONTINUE WITH THE NEXT ITERATION."

The format of continue statement is

```
        Continue;
e.g.    While (test-condition)
        {
                ........
                if (condition)
                continue;
                ......
        }
```

The **continue** statement is useful in loops and will perform the similar function but the control is transferred back to test the condition of the loop i.e. it jumps you out of the current iteration of a loop. Remember that break jumps completely out of a loop.

Thus, you use break when you want to jump out of loop and terminate it but if you don't want to terminate loop and simply want to jump immediately to the next iteration of the loop, use continue.

EXERCISE

1. What is a variable?
   (a) A value that can not be changed.
   (b) Text in a program that the compiler ignores.
   (c) A place to store information in a program.
2. What is the process of fixing errors called?
   (a) Defrosting
   (b) Debugging
   (c) Decomposing
3. State whether the following statements are TRUE and FALSE:
   (a) Every line in a Java Program should end with a semicolon.
   (b) In Java language, lowercase letters are significant.
   (c) Every Java program ends with an END word.
   (d) main () is where the program begins its execution.
   (e) A line in a Java program may have more than one statement.
   (f) A System.out.println( ) statement can generate only one line of output.
   (g) The closing brace of the main() in a program is the logical end of the program.
   (h) Syntax errors will be detected by the compiler.
   (i) All variables must be declared before their use in a program.
   (j) An integer variable will be initialized to zero automatically if not done explicitly.
   (k) Comments must be restricted to one line only.
   (l) Comments must begin at the start of a line only.
   (m) In nested if statement, the last else gets associated with the nearest if without an else.
   (n) One if can have more than one else clause.
   (o) You can always replace a switch statements by a series of if..else statements.
   (p) A switch expression can be of any type.
   (q) A program stops its execution when a break statement is encountered.
   (r) A execution of a part of loop can't be skipped.
4. If semicolons are needed at the end of each statement, why does the comment line starting with // goes here not end with a semicolon?
5. What is the major advantage of using /** **/ for comments?
6. Discuss the various basic datatypes.
7. What is a variable and what is meant by the "value" of a variable? What is initialization? Why is it important?

8. What is the significance of Block and Scope?

9. Differentiate between "break" and "continue" giving examples.

10. What is the difference between an **if** and a **switch** statement.

11. Find the sum of all odd no's between 1 and 100. Try to use different looping techniques.

12. Write a program to print the Floyd's triangle shown below:

```
1
2    3
4    5    6
7    8    9    10
11   12   13   14   15.....
79.....91
```

13. Write a program using do....while loop to calculate & print n numbers of Fibonacci series:

    1 1 2 3 5 8 13 21.....

    (From third number in series, the number is sum of previous two numbers)

14. Compare in terms of their functions, the following pairs of statements:

    (a) while and do ..while

    (b) while and for.

    (d) break and continue.

OBJECT ORIENTED PROGRAMMING

JAVA CLASS
OBJECT CREATION
INHERITANCE
CASTING
INTERFACE
PACKAGES
SOME JAVA PACKAGES

Object Oriented Programming

As a software developer, you will always look out for the approaches that will help you develop quickest, cheapest and reliable softwares. There are many approaches and technologies that have evolved since the dawn of computing. The reason for having many approaches can be contributed to improvements in hardware technology, new innovations in computer science and changes in user expectations over period of time.

Of all the evolved approaches, the Object- Oriented approach has proven itself to be the best approach for a large class of software. It may be possible that a new approach may replace the existing approach but till now the Object-Oriented approach is the best approach for the majority of software that you develop today. This approach is mainly focused on the development of self-contained software components, called objects.

OBJECTS

To understand the concepts of Objects, just look around your self to locate real-world objects. Cars, TV, Motorbike, computers, pen etc. are all objects. Let us pick computers and understand various aspect of this object.

Your computer is an **Object** contains a lot of information. It also has **methods** for accessing the information. You have to switch on the computer, browse files using different softwares, search the files, read a file, search the directory etc. You could access the information contained in the computer, using various methods for accessing it and this makes this computer an Object.

Next aspect associated with the object is - its **state**. The state of an object is the particular condition it is in. For example, a computer can be on or off. The methods-turn computer on and turn computer off- are used to access the state of the computer. If the computer is on, it could have a file open or closed. This is also another state of computer. The files are objects in their own right. They contain information and can be accessed through the proper software. Thus software is a method for file.

The computer object can be viewed as being composed of file objects. The computer's methods provide access to files, and the files methods provide access to the information contained in a particular file.

The information contained in an object, whether it is state specific or not, is referred to as the object's **data** e.g. the files contained in the computer is data. You can access the data using Object methods. The method could provide the information supporting **read access,** whereas other methods could let you modify the data supporting **write access**. Finally, you could also have the methods which will let you create objects, called **constructors**.

You can compose or built objects from other objects. This is one of the major advantage of object-oriented programming, which helps you to create complicated objects using the simple objects. This way you can develop a complicated application using simple objects.

Another advantage of object-oriented concept is that you can reuse the object, created for one application, for another application. With this facility, you have the capability to build or acquire a library of objects from which you can more quickly and easily piece together your programs. For example, if you are developing a game, you can develop characters, backgrounds and many other items as object. Same objects you can use to create in different other games. This reduces your work for a new game as you don't have to start from scratch.

CLASS

We have just discussed, how an object can simplify the process of development. But how to develop objects. In object-oriented approach, the most fundamental structure is class, which we use to develop objects. A class is a template or prototype that defines a type of data that is contained in an object and the methods that are used to access this data. Using the same class, you can develop as many objects as you want.

A class is same to an objects what a drawing to a machine component. Using the same drawing, you can develop as many components as you want and all the components will share common characteristics. All components have states and behaviors in common that define them as component. When engineers start creating a new component, they will typically build them all from drawing. It wouldn't be as efficient to create a new drawing for every single component, especially when there are so many similarities shared between each one.

Similarly, in object-oriented programming, it's also common to have many objects of the same kind that share similar characteristics. And like the drawing for similar components, you can create drawings for objects that share certain characteristics and these drawings are Classes for software development.

Thus in other words, objects which are similar in structure and other attributes, are grouped together. This group is called Class. In classes we have parent class, subclass, and superclass which allows us the level of abstraction for grouping objects.

INHERITANCE

Its another advantage of object oriented database. Inheritance means to inherit. When you define a class, it has some property and attributes associated with it. Now you need another class which contains all the properties and attributes of this class and few additional attributes. You can define the new objects in terms of existing class known as base class. All the objects created in lower-

level of the base class will inherit the properties and attributes of all classes above them.

Thus inheritance will organise the classes in a hierarchical fashion. Consider a class Y which is an extension of another class X if it contains all the data contained in class X and implements all the methods implemented by class X. Thus Class Y will be a subclass to class X and class X will be a superclass or parent class of class Y.

Classes form a hierarchical classification tree under the subclass relationship. If a class Y is a subclass of a class X, it inherits the properties of X. This means that all of the data and methods defined for class X are available to class Y.

MESSAGE AND METHOD

Object working alone are useless. For example, a computer as an object is useless piece unless an operator starts it, runs various instructions and achieves the objective. Here operator is another object and it sends message to computer to perform certain functions.

Thus objects must interact with each other to bring out something useful. At times, this interaction is accompanied by various parameters. For example operator has to open a file on the computer and it also has to tell that the file should be opened in particular software. Thus the particular software is parameter for opening a file. This means that objects anywhere in a system can communicate with other objects solely through messages accompanied by parameters.

In object oriented programming, when an object is sending a message to another object, it is, in fact calling the method of the other object. For example, when an operator asks the computer to open a file using a particular software, he is actually calling method of opening the file. The message parameters are actually the parameters to a method. In object- oriented programming, messages and methods are synonymous.

ENCAPSULATION

After defining Class and Object, the data is bound to the object so that access to the object can happen only through the ways that has been defined for operating on it. This process of packaging an object's data together with its methods is called encapsulation. A powerful benefit of encapsulation is the hiding of implementation details from other objects. This gives a better security against illegal access.

After encapsulation, an object is provided with a public interface to interact with other objects. The private section of the object can be a combination of internal data and methods. The internal data and methods are the sections of the object hidden.

JAVA CLASS

All the classes in Java fan out from the Object base class. Object serves as the superclass for all derived classes, including the classes that make up the Java API.

Now, as we are well versed with the object oriented approach, let us consider the Order Processing System for publisher. In this system, we have items as books having title, number of page, rate, language, quantity in stock and reorder level. These are the various attributes important for Order Processing System.

Here books are objects and various methods must be identified for the objects. The methods could be stock status, stock value, reordering or reaching at reorder level, receiving stock etc. These are all the methods which could be applied on the object in the system.

DECLARING CLASSES

To start with, we have to create the object. In OOPs, objects are created by using a class of objects as a guideline. Therefore, we must declare the class before creating the object. The syntax for declaring classes in Java follows:

```
class Identifier {
      ClassBody
}
```

Here, identifier specifies the name of the new class. The curly braces surround the body of the class, classbody.

Consider the class book in Order Processing System for Publisher.

```
class Book {
    private string title ;
    private int page;
    private float rate;
    private string lang;
    private int quantity;
    private int re_q;
}
```

(You may note the string type data which is actually not a data type but a class defined in package java.lang. It will be discussed in more details in coming chapter.)

DECLARING METHODS

The state of the book object is defined by six data members. Now book class itself is not useful, it also needs some methods. The most basic syntax for declaring methods for a class follows:

```
ReturnType Identifier(Parameters) {
    MethodBody
}
```

Here, Returntype specifies the data type that the method returns, Identifier specifies the name of the method, and Parameters specifies the parameters required to the method. The parameters are optional. You also have MethodBody enclosed by curly braces. As we have already discussed that a method is synonymous with a message in OOPs. Consider the following method for reordering the books:

```
void Reorder(int re_q, int quantity) {
if (quantity < re_q) {
    // Reorder the book
    }
else {
    // Enough stock, no need to reorder
    }
}
```

The Reorder() method is passed with two integers, re_q and quantity. These value are then used to determine whether the books should be ordered or not. If you make the Reorder() method a member of the Book class, then you need not have to give the parameters with the method. This is because both the parameters are already member variable of Book, to which all class methods have access. The Book class, with the addition of the Reorder() method, will look like as given below:

```
class Book {
        private string title ;
        private int page;
        private float rate;
        private string lang;
        private int quantity;
        private int re_q;
    public void Reorder() {
    if (quantity < re_q) {
        // Reorder the book
        }
    else {
        // Enough stock, no need to reorder
        }
    }
    public void Display(){
        System.out.println("Book Details");
        System.out.println("————————————");
        System.out.println("Book Title : "+title);
```

```
        System.out.println("Numer of pages "+ page);
        System.out.println("Price : "+ rate);
        System.out.println("Quantity in hand : "+ quantity);
        System.out.println("Language : "+ lang);
        System.out.println("Reorder Level  : "+ re_q);
    }
    }
```

CONTROLLING THE ACCESS

The above class Book defines six variables and two methods - Reorder() and Display(). The variables are defined as private, whereas methods are defined as public. These definitions - private, public etc. controls the access to class and attributes and are called Access modifiers. Access modifiers define varying levels of access between class members and the outside world (other objects). These are declared immediately before the type of a member variable or the return type of a method.

If no access modifier is defined before the variable or method, the class and variables can be accessed by other classes in the same package.

Private V/s Public Access Modifiers

If any attribute or method of a class is declared as public, it will be available to all the other classes, but if an attribute or a method of a class is declared as private, it will not be available to any other class except for the class in which it is declared.

Here, in the above example, methods have been declared public so that the data of the class could be accessed only by using methods.

Protected Access Modifier

When you add **protected** before the type of a member variable or the return type of a method, it specifies that class members are accessible only to methods in that class and subclasses of that class.

```
class Book {
        protected string title ;
        protected int page;
        .....
        }
```

Static Access Modifier

When you require any common variable or method for all the objects of a class, you may define it as static variable or method.

```
Static string title ;
Static void Reorder() {
}
```

Final Access Modifier

The final modifier specifies that a variable has a constant value or that a method cannot be overridden in a subclass (Overriding a class will be discussed later in the chapter).

```
final public int price = 25;
```

OBJECT CREATION

Once you have created the classes, you need to use the classes in the program. You can create the object using special method called **constructor**. The constructor method allows you to initialize variables and perform any other operations when an object is created from the class. You can also implement constructor in all your classes. The constructor is always given the same name as class.

Consider the following class where two constructor are provided:

```
class Book {
    private string title ;
    private int page;
    private float rate;
    private string lang;
    private int quantity;
    private int re_q;
public Book() {
    private string title = "ITC" ;
    private int page = 372;
    private float rate = 99;
    private string lang = "English";
    private int quantity = 650;
    private int re_q = 200;
    }
public Book(string a, int b, int c, String d, int e, int f) {
    string title = a ;
    int page = b;
    float rate = c;
    string lang = d;
    int quantity =e;
    int re_q = f;
    }
public void Reorder() {
if (quantity < re_q) {
    // Reorder the book
    }
```

```
        else {
            // Enough stock, no need to reorder
            }
        }
        public void Display(){
            System.out.println("Book Details");
            System.out.println("——————————");
            System.out.println("Book Title : "+title);
            System.out.println("Numer of pages "+ page);
            System.out.println("Price : "+ rate);
            System.out.println("Quantity in hand : "+ quantity);
            System.out.println("Language : "+ lang);
            System.out.println("Reorder Level  : "+ re_q);
        }
        }
```

Here, first constructor takes no parameters and initializes the member variables to default values. The second constructor takes all the six values and initializes the member variables with them.

Now, you can create an instance of a class by declaring an object variable and using the **new** operator.

```
Book book1 = new Book();
Book book2;
book2 = new Book(string a, int b, int c, int d, int e, int f)
```

The above commands create two objects book1 and book2 of the class Book. In these command, **new** operator creates the objects, i.e., it allocates memory for the class's data. You may also note that name of both the constructor is same. The first constructor doesn't take any attribute, but the second constructor enables you to specify different types of information (parameters) to send to the attributes. This is called **constructor overloading**. Now how the compiler will decide, which constructor has been called. A constructor in Java is recognised by signature consisting of name, number of arguments it accepts and argument type. When a call to constructor is made, Java matches the signature of the function that calls the constructor to the signature of the constructor declared in the class. Then it calls the constructor that has the same signature as the calling function.

Consider the following code where constructor is called in another class called Sample. To use the class and constructor Book, it must be imported here using the Import statement in the beginning of the code:

```
import Book;
class Sample{
    public static void main(String args[]){
// displays the contructor1 details
```

```
        Book  book1=new Book();
        book1.Display();
    // display  Constructor 2 details
    Book book2 = new Book(String "Java2", int 240, int
    99, String "English", int 700, int 200);
    book2.Display();
// Gives Error
    Book book3=new Book(String "Java2", int 240);
    book3.Display();
}
}
```

The above code displays the book1 and book2 but gives error in book3 as no constructor having two attributes, is defined in the class Book. You may also note the usage of dot(.) notation. which is used to call the method of a class. e.g. book2.display() calls the method display for the object instance book2 of class Book. It will display the contents of book2 using display() method.

INHERITANCE

Till now, we have discussed only about creating new classes and creating instance from the classes. Deriving all your classes from Object isn't a very good practice as you would have to redefine the data and methods for each class. To reduce the workload, you can derive subclasses from classes using the extends keyword.

The subclass inherits all the properties of class and it could be given additional properties. The syntax for deriving a class using the extends keyword follows:

```
class Identifier extends SuperClass {
    ClassBody
}
```

Identifier refers to the name of the new subclass, SuperClass refers to the name of the class you are deriving from, and ClassBody is the new subclass body.

Consider a subclass - CompBook of Book. It contains all the books related to computers. This class has all the attribute that any book has along with two additional attribute namely **softw** and **Auth**. The two attributes are of string type and tells about software on which book is written and authors name respectively. Following is the Book-derived Compbook class using the extends keyword:

```
class Compbook extends Book{
    String softw;
    String Auth;
public Compbook() {
    super();
```

```
        String softw  =”Flash 5”
        String Auth = “M.N.Gulati”
        }
    public Compbook(string a, int b, int c, String d, int e,
    int f, string g, string h) {
        super (a,b,c,d,e,f);
        this.softw = g;
        this.Auth = h;
        }
```

This declaration assumes that the Book class declaration is readily available in the same package as Compubook. You can also derive a class from an external superclass. To derive a class from an external superclass, you must first import the superclass using the import statement as discussed earlier.

e.g. **import Book;**

ABSTRACT CLASS

At times, you can create a superclass which act purely as templates for more useable subclasses. In such a situation, superclass serves as design templates for other classes. These types of superclasses are referred to as abstract classes and cannot be used to create objects.

The abstract class is a class, whose purpose is solely a design convenience. Abstract classes are made up of one or more abstract methods, which are methods that are declared.

In above example, the Book class can be declared as abstract class and similar to Compbook class, you can create other classes. To create abstract class, you can use the abstract keyword:

```
abstract class Book{
    ..................
    abstract Display()
    ..................
    abstract Reorder()
    ..................
}
```

Here, abstract keyword is used before the class declaration, which indicates that this class can’t be used to declare instance. Also, the methods defined in the class has also been declared abstract. You may also note that

- can’t make constructors abstract.
- can’t declare static and private method as abstract.

OVERRIDING METHODS

You can override methods in the derived classes. For example, the Display() method in the Book, when used with the Compbook class,

will display the incomplete information. Therefore, for any instance of class Compbook, the Display()method must be redefined. To incorporate the additional information of Compbook class, you would override the Display() method with a version specific to Compbook class. The Compbook class would then look something like this:

```
class Compbook extends Book{
        String softw;
        String Auth;
    public Compbook() {
        super();
        String softw  =”Flash 5”
        String Auth = “M.N.Gulati”
        }
    public Compbook(string a, int b, int c, String d, int e,
    int f, string g, string h) {
        super (a,b,c,d,e,f);
        this.softw = g;
        this.Auth = h;
    public void Display(){
        System.out.println(“Book Details”);
        System.out.println(“———————”);
        System.out.println(“Book Title : “+title);
        System.out.println(“Numer of pages “+ page);
        System.out.println(“Price : “+ rate);
        System.out.println(“Quantity in hand : “+ quantity);
        System.out.println(“Language : “+ lang);
        System.out.println(“Reorder Level  : “+ re_q);
        System.out.println(“Software : “+ softw);
        System.out.println(“Author : “+ Auth);
        }
        }
```

Now, create an instance of the Compbook class as shown below:

```
Compbook Comp1 = new Compbook();
```

If this instance call the Display() method i.e. Compbook.Display(), the new Display() method in Compubook is executed rather than the original overridden Display() method in Book.

OVERLOADING METHODS

Similar to constructor overloading, you can overload the methods also. In method overloading also, you send different types of information (parameters) to send to a method. You can overload a method by declaring another method with the same name but different parameters.

For example, the Reorder() method for the Book class could have two different versions: one for general reorder by comparing the quantity with reorder level and other for ordering the book if the quantity of the book is less than a certain percentage of the reorder-level. (If the reorder level is 200 books, you might like to place the order when the stock is 300 books or 100 books).

The declaration of the general version is one you've already defined:

```
public void Reorder() {
if (quantity < re_q) {
    // Reorder the book
    }
else {
    // Enough stock, no need to reorder
    }
```

To enable the Reorder() method to calculate the certain percentage of the stock, you might pass a variable (say x) for calculating that value.

```
public void Reorder(int x) {
if (quantity < re_q*x) {
    // Reorder the book
    }
else {
    // Enough stock, no need to reorder
    }
```

Notice that the only difference between the two methods is the parameter lists; the first Reorder() method takes no parameters; the second Reorder() method takes one integer.

Again, similar to constructor, compiler keeps up with the parameters for each method along with the name. When a call to a method is encountered in a program, the compiler checks the name and the parameters to determine which method to be called.

CASTING

We have already discussed the casting of datatypes. As you know, casting is a process of changing the value so that it could fit itself into new datatype. What we discussed earlier was casting between primitive data types *(int, float, boolean*, etc.). With the introduction to classes and object, casting can be applied between objects *(String, Point, Integer*, etc.) or between a primitive data type and an object.

The casting between data value of primitive data types simply converts a data value of a primitive data type to another primitive data type. While casting, care should be taken that the destination type should always be equal to or larger in size than the source type so that there is no loss of information. However, if the destination cannot hold

larger value than the source, explicit casting must be done. Explicit casting is done in the following way:

```
(data type)value
```

For example, the following set of commands converts a *float* value to an *int* value.

```
.............
int i;
float f=6.7;
i=(int)f;
```

Casting Objects

You can also convert an object of a class to object of another class. But the condition is that both these classes must be related to each other by inheritance, i.e., one class must be a subclass or superclass of the other class. You can divide the casting between classes into two different situations:

- Casting from a subclass to a superclass
- Casting from a superclass to a subclass

Here also casting can either be implicit or explicit. In the case of casting from a subclass to a superclass, you can cast either implicitly or explicitly.

Implicit casting happens when an object of smaller type is casted to an object of larger type and for implicit casting you do nothing. Thus you can use an instance of subclass instead of instance of superclass without explicit casting. This is because all the attributes and methods of a super class are automatically available to all of its subclasses.

For example, and function that accepts the Book class will also accept the Compbook class as Compbook class contains all the attributes and methods of Book class.

Explicit casting is done when an object of subclass has to be casted to an object of super class. You must remember that explicit casting results in loss of information, as we have seen in the case of datatypes. Here in case of class, when a subclass will will be casted to superclass, the additional data of the subclass will be lost. For explicit casting you can use the following operation:

```
(classname)object
```

Consider the following set of instructions:

```
Book book1 = new Book();
Compbook CB1 = new Compbook();
book1 = CB1;                    // valid statement
CB1 = book1;                    // invalid statement
CB1 = Compbook(book1)           // valid statement
```

Casting of an object to datatype or viceversa is not allowed in Java. For such a purpose, Java has provided its users with an alternative package (java.labg) include classes that correspond to primitive data type. This will be discussed in detail in coming chapters.

OBJECT DESTRUCTION

In Java, each object of a class is allocated memory for its individual attributes, but all the objects of the same class will share the same methods. i.e. methods will be stored at one place and all objects will use methods from there.

Also the memory management in Java is automatic and you don't have to deallocate the memory. When an object falls out of scope, it is automatically removed from memory, or deleted. This is done by garbage collector by reclaiming the memory of the objects that are no longer in use.

But at times it is required to release the resources that may be destroyed with the object. For such purpose, Java provides the ability to define a destructor that is called just before when an object is deleted. The destructor is called **finalize()** and process of defining the finalize() method is called finalization.

```
void finalize() {
    // cleanup
}
```

You may note that finalize() method will be invoked only when garbage collector in invoked, which occurs at inconsistent intervals. Therefore, the critical information must not be retrieved using finalize() method.

INTERFACE

An interface is a prototype for a class and it consist of only the constants and the method prototypes. It is similar to the abstract class which are declared as design templates for further classes. The interfaces are different from abstract classes in the implementation. In abstract classes, the classes are left partially unimplemented as it uses abstract methods which are themselves unimplemented.

You may consider interfaces as abstract classes that are left completely unimplemented. Completely unimplemented means that no methods in the class have been implemented. Also the interface member data should be static limited variable i.e. constant.

To understand the interface let us consider an example. Suppose, we have two classes Magzine class and Textbook class. Now these are the two classes that have one thing in common - paper. Both the class will use paper, so we define an interface - paper interface which these two classes will have to implement. Now it is upto the class that what quality of paper will be used by the class.

The major benefits of using interface are

- You can define the protocol without worrying about the implementation. Once interfaces have been designed, the class development can take place without worrying about communication among classes.
- It provides a substitute for multiple inheritance. (Multiple inheritance is not allowed in Java). The benefit of using the interface approach is that it enables you to inherit only method descriptions, not implementations.

DECLARING INTERFACES

You will use the following syntax for creating interfaces:

```
interface Identifier {
    InterfaceBody
}
```

Identifier specifies the name of the interface and InterfaceBody refers to the abstract methods and static final variables that make up the interface. As all the methods in an interface are abstract, you need not to use the abstract keyword.

IMPLEMENTING INTERFACES

Implementing an interface is similar to deriving from a class, except that you use implements keyword instead of extends keyword. The syntax for implementing a class from an interface follows:

```
class <class-name> implements <interface-name>{
    //instance-variables
    //class-methods
    //interface-methods
}
```

class-name specifies the name of the new class. Interface-name is the name of the interface you are implementing, and in the body, you will define instance variable, class-method and interface method.

Consider the following example, where sales order are being processed by the publisher for general books and computer books. The method of calculating the net order value has following components:

Quantity
Price
Normal Discount
Quantity Discount
Cash Discount
Trade Tax

Using the above variable, bills is to be generated for various parties, who purchases, books or magzines. Both classes require the methods

for the calculation of Normal Discount, Cash Discount, Quantity Discount, Trade Tax and Net Amount. So here you can declare an interface called Discount() which is defining these method prototypes and a publisher name as a constant, since the publisher name has to be used identically in both the classes.

```
// Declaring Interface
interface Discount
      {
        final static string publisher = "SiliconMedia";
        public float Tot_Cost();
        public float Normal_Dis();
        public float Cash_Dis();
        public float Qty_Dis();
        public float  Trade_Tax();
        }
// Declaring the Class Book
class Book{
        protected int ISBN;
        String party;
        String title;
        protected int price;
        protected int qty;
        protected int total_cost;
        int paymode;
        Book(int bisbn, string bparty, string btitle, int bprice, int
        bqty, char bpaymode)
        {
            ISBN = bisbn;
            party = bparty;
            title = btitle;
            total_cost =  bprice * bqty;
            paymode = bpaymode;
            qty = bqty;
            price = bprice;
        }
        }
// Declaring the Class Magzine derived from Class Book and
//implementing interface Discount
class Magzine extends Book implements Discount {
        Magzine(int misbn, string mbparty, string mtitle, int
        mprice, int mqty, int mpaymode)
            {
            super(misbn, mparty, mtitle, mprice, mqty,
            mpaymode);
```

```
    }
public float  Normal_Dis() {
    if (total_cost > 25000)
          return ((float) 0.15 * total_cost);
    else
          return ((float) 0.10 * total_cost);
    }
public float  Cash_Dis() {
    if (paymode > 1)
          return 0;
    else
          return ((float) 0.02 * total_cost);
    }
public float  Qty_Dis() {
    if (qty > 1000)
          return ((float) 0.03 * total_cost);
    else
           return 0;
    }
public float  Trade_Tax() {
    float Gros_Cost = total_cost - Normal_Dis() - Qty_Dis()
    - Cash_Dis();
    return ((float) 0.04 * Gros_Cost);
    }
public float  Tot_Cost() {
    return (total_cost - Normal_Dis() - Qty_Dis() - Cash_Dis()
    +Trade_Tax());
    }
void print_Bill()
    {
    System.out.println(Discount.publisher + "\n " );
    System.out.println("ISBN : " + ISBN);
    System.out.println("Title : " + title);
    System.out.println("Purchaser : " + party);
    System.out.println("Quantity :  " + qty);
    System.out.println("Price : " + price);
    System.out.println("Normal Discount " + Normal_Dis());
    System.out.println("Quantity Discount :" + Qty_Dis());
    System.out.println("Cash Discount :" + Cash_Dis());
    System.out.println("Trade Tax  :" +Trade_Tax());
    System.out.println("Net Cost : " + Tot_Cost());
    System.out.println("_____________\n");
    }
```

```
        }
// Declaring the Class Text book derived from Class Book and
//implementing interface Discount
class Textbook extends Book implements Discount{
        Textbook(int misbn, string mparty, string mtitle, int
        mprice, int mqty, int mpaymode)
            {
            super(misbn, mparty, mtitle, mprice, mqty,
            mpaymode);
            }
        public float  Normal_Dis() {
            if (total_cost > 50000)
                  return ((float) 0.20 * total_cost);
            else
                  return ((float) 0.10 * total_cost);
            }
        public float  Cash_Dis() {
            if (paymode > 1)
                  return 0;
            else
                  return ((float) 0.03 * total_cost);
            }
        public float  Qty_Dis() {
            if (qty > 1000)
                  return ((float) 0.02 * total_cost);
            else
                   return 0;
            }
        public float  Trade_Tax() {
            float Gros_Cost = total_cost - Normal_Dis() - Qty_Dis()
            - Cash_Dis();
            return ((float) 0.01* Gros_Cost);
            }
        public float  Tot_Cost() {
            return (total_cost - Normal_Dis() - Qty_Dis() - Cash_Dis()
            + Trade_Tax());
            }
        void print_Bill()
            {
            System.out.println(Discount.publisher + "\n " );
            System.out.println("ISBN : " + ISBN);
            System.out.println("Title : " + title);
            System.out.println("Purchaser : " + party);
            System.out.println("Quantity :  " + qty);
```

```
            System.out.println("Price : " + price);
            System.out.println("Normal Discount " + Normal_Dis());
            System.out.println("Quantity Discount :" + Qty_Dis());
            System.out.println("Cash Discount :" + Cash_Dis());
            System.out.println("Trade Tax  :" +Trade_Tax());
            System.out.println("Net Cost : " + Tot_Cost());
            System.out.println("_____________\n");
            }
            }
public class calc{
            public static void main(String args[ ]){
            Magzine M1 = new Magzine(1097, "RGC", "MediaQuest",
            100, 500, 1);
            M1.print_Bill();
           Textbook T1 = new Textbook(1001, "RGC", "SAD", 75,
            1000, 1);
            T1.print_Bill();
            }
        }
```

The output of the above calc.java would be as follows:

```
ISBN : 1097
Title : MediaQuest
Purchaser : RGC
Quantity :  500
Price : 100
Normal Discount 7500.0005
Quantity Discount  :0.0
Cash Discount :1000.0
Trade Tax  :1660.0
Net Cost : 43160.0
_____________
ISBN : 1001
Title : SAD
Purchaser : RGC
Quantity :  1000
Price : 75
Normal Discount 15000.0
Quantity Discount  :0.0
Cash Discount :2250.0
Trade Tax  :577.5
Net Cost : 58327.5
_____________
```

PACKAGES

You can group related classes and interfaces together into a single unit called package. When you are handling a large group of classes and interface, it makes you job easier if you group together the classes and interface.

The concept of package can be well compared to the concept of files and folders. Imagine a scenario, where your all the files are in a single folder. It will makes you task a hell, of locating a file. The folder concept lets you easily locate you files. Package are similar to folders, which will help you locating the various classes and interface.

There is another benefit of placing the classes into package. By placing classes into a package, you allow classes in the same package to access each other's class information.

DECLARING PACKAGES

You use the following syntax for the package statement:

package Identifier;

You can place the above statement at the beginning of a compilation unit, before any class declaration. Every class located in a compilation unit with a package statement is considered part of that package.

The steps involved for creating the package are:

Create the directory structure as drive:\jdk1.2.2\Book. The folder Book contains the file declaring the package. The first statement of such file (say Newbook.java) should be

package Newbook;

In the class file, which declares the package, there must be only one class with the public identifier which is the class of the same name as the filename. Rest all classes can be with private modifier etc.

```
public class Newbook
    {
            .....................
    )
```

Here, Newbook is a package (storing Newbook.class file) belonging to package Book. The package hierarchy automatically gets created with the directory structure.

You can also place the packages into other packages. While doing so, you must follow the directory structure containing the executable classes to match the package hierarchy.

IMPORTING PACKAGES

You can also use the classes from the other packages by using import statement. This statement lets you to import classes from other packages into a compilation unit. The syntax for the import statement follows:

import Identifier;

Here, identifier is the name of the class or package of classes you are importing. For example, the following statement will import all classes of java.lang package.

import java.lang.*;

If you want to import a particular class from the package, you could specify the class name.

import java.lang.String

The above statement will import the String class from Java.lang package.

SOME JAVA PACKAGES

Java comes with certain built-in packages that perform all important tasks. These include:

- **Java.lang** package contains all the fundamental classes like the Security Manager class, System class, Thread class etc.
- **Java.util** package contains the utility classes like the Date class etc.
- **Java.io** consist of the classes responsible for the input output operations, like the disk I/O etc.
- **Java.applet** carries the necessary functionality for creating and managing the applets.
- **Java.awt** contains the GUI components meant for the interface designing, event handling etc., like the Button class, Label class etc.
- **Java.net** package contains the classes for operations over the net like establishing connection to a remote server etc.

EXERCISE

1. How did object-oriented programming get its name?
   (a) Programs are considered to be a group of objects working together.
   (b) People often object because it's hard to master.
   (c) Its parents named it.
2. What is a program's capability to handle more than one task called?
   (a) Synchronization
   (b) Multiculturalism
   (c) Multithreading
3. What statement is used to enable one class to inherit from another class?
   (a) inherits
   (b) extends
   (c) handitover
4. Why are compiled Java programs saved with the .class file extension?
   (a) Java's developers think it's a classy language.
   (b) It's a subtle tribute to the world's teachers.
   (c) Every Java program is a class.
5. What are the two things that make up an object?
   (a) attributes and behavior
   (b) commands and comments
   (c) variable and comments
6. What is a method an example of in a Java class?
   (a) attributes
   (b) statements
   (c) behavior
7. If you want to make a variable a class variable, what statement must you use when it's created?
   (a) new
   (b) public
   (c) static
8. Can classes inherit from more than one class?
9. Why are object-oriented programs easier to debug?
10. When would you want to create a class that isn't public?
11. Fill in the Blanks:
    1. Objects are _____ of classes.
    2. _____ operate on data in the class.
    3. The methods that have the same name as the class are called _____.

4. Objects are instantiated using _____ operator.
5. The getMethod() is a method of _____ class.

12. Do you have to create an object to use class variables or methods?
13. What does it mean that a class or member is “final”?
14. What does it mean that a method or field is “static”?
15. What does it mean that a method or class is abstract?
16. What’s an interface?
17. How does garbage collection work?
18. What is package? What are the main standard packages in Java?

MULTITHREADING

INTRODUCTION
THREAD
JAVA THREADS
CREATING THREADS
THREAD GROUPS
SYNCHRONIZATION

Multithreading

INTRODUCTION

Multithreading - as the name suggest - is more than one thread. This thread is thread on control within a program. You can think of thread as a subprogram within a program. Thus, Multithreading means controlling more than one subprograms at a time. As Multitasking is the property of Operating System, which lets you run more than one program at a time, Multithreading is the property of program, which lets you run more than one sub-program with in a program, at a time.

Multithreading is an integral part of Java language and this factor distinguishes Java from other programming languages as most of the other modern programming languages either do not offer multithreading or provide multithreading as a nonintegrated package.

If you have knowledge about MS-DOS (Disk Operating System), you must be aware that it is single tasking operating system, i.e. you must finish one job to start another job. The scenario was extremely frustrating at that time as the computer would start one job, run that job to completion, then start the next job, and so on. The urge to work faster on the computers, generated the requirement of running more than one job at a time and thus was born the modern multitasking operating system.

All the modern operating system provides you the capability of performing multiple jobs concurrently. You can type a letter in wordprocessor, design a balance sheet, download the message from net, and edit your photograph and browse your computer - all at the same time. You don't have to quit one job to perform another.

THREAD

A thread is a sequence of instructions in a program. So every program or application you design is a atleast a single-threaded application. Now when we say multithreading, we intend to run more than one sequence of program - thread - at a time.

Now, how a single CPU is able to execute more than one thread at a time. In single-processor systems, only a single thread of execution occurs at a given instant. The CPU moves back and forth among various threads to create the illusion that threads are being processed at a time. Single-processor systems support logical concurrency and not physical concurrency. Logical concurrency occurs when multiple threads execute with separate, independent flows of control. The Physical concurrency occurs when the threads are actually executed at the same time. This happens only in multiprocessor system. The important feature of multithreaded programs is that they support logical concurrency irrespective of the fact that whether physical concurrency is actually achieved.

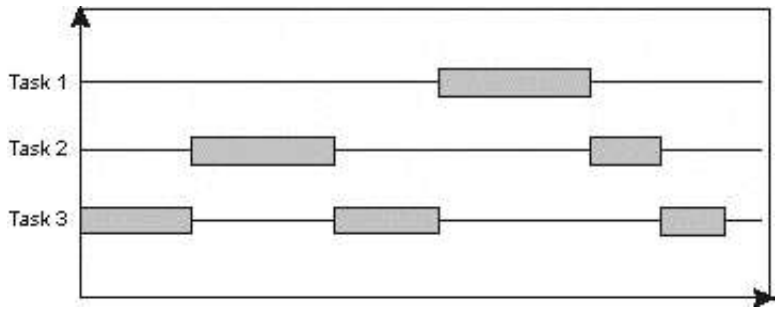


Fig 4.1 Multithreading on a single processor system

JAVA THREADS

Java multithreading has been built around the java.lang.Thread class. The **Thread** class provides the capability to create objects of class **Thread**, each with its own separate flow of control. This class encapsulates the data and methods associated with separate threads of execution and allows multithreading to be integrated within the object-oriented framework.

Also, all class libraries have been designed keeping multithreading in mind i.e. if a thread stops executing for some time, rest of the program is not affected.

Thread States

A thread has a life span and during the life it passes through five stages: new, ready, running, inactive and finished. As soon as thread is born or created, it enters the new state. When started by start() method, it is ready to run. When start() method calls the run() method, the thread comes in running state.

However, the thread may not always be in the running state. It may become inactive for one or more reason. When a thread is inactive, it implies that the thread is alive and that it can be allocated CPU time by the system for execution. Some of the events that may cause a thread to become inactive include the following:

- Time not allotted by CPU in single processor system.
- The thread has been put to sleep for a certain period of time using the sleep() method.
- The thread is waiting for execution as wait() method has been called.
- The thread has been suspended using the suspend() method.

A thread that is inactive may either be resumed, in which case it enters the ready state again, or it can be stopped in which case it enters the finished state.

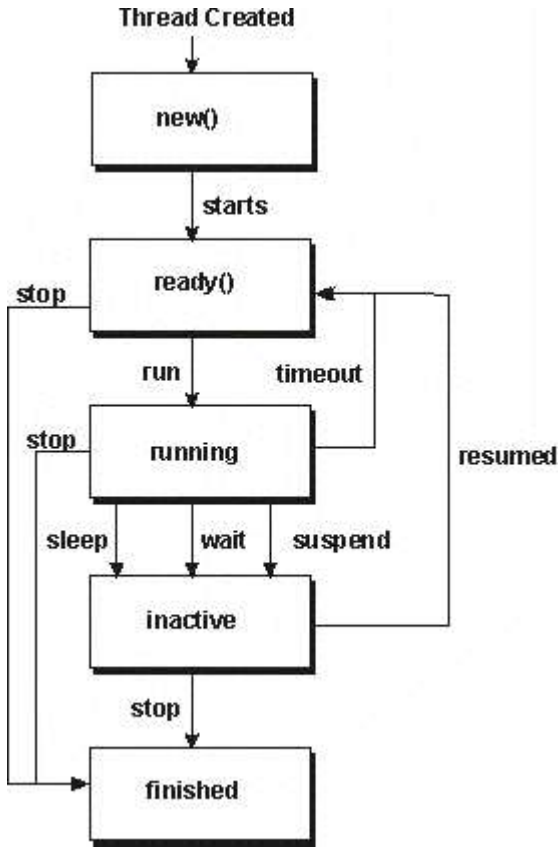


Fig 4.2 Various states of thread

CREATING THREADS

The thread creating process involves two steps - writing the code that is executed in the thread and writing the code that starts the thread. In all the program developed so far, you have used single thread. When you start a Java application, the virtual machine (VM) runs the main() method inside a Java thread. The Java virtual machine provides a multithreaded environment, but it starts user applications by calling main() in a single thread.

To create threads, Java provides two approaches: by extending Thread class and by implementing Runnable interface.

In the first approach, you create a subclass of class Thread by extending Thread class and override the run() method to provide an entry point into the thread's execution. This method implements the main logic of the thread. When you create an instance of your Thread subclass, you invoke its start() method to cause the thread to execute as an independent sequence of instructions. The start() method is inherited from the Thread class. It initializes the Thread object using

your operating system's multithreading capabilities and invokes the run() method.

This method limits the use of run() method as the Thread objects to be under the Thread class in the class hierarchy. At many a times, you may want to add a run() method to a preexisting class that does not inherit from Thread.

In such cases, you can implement java.lang.Runnable interface. The Runnable interface consists of a single method, the run() method, which must be overridden by your class. The run() method provides an entry point into your thread's execution. In order to run an object of your class as an independent thread, you pass it as an argument to a constructor of class Thread.

CREATING THREAD SUBCLASS

You can create a new thread by subclassing java.lang.Thread. Todoso, you create a subclass of Thread and then create, initialize and start two Thread objects from your class. The two threads will execute concurrently to display the output. Lets consider that both the threads has to display the message "This book on Java is quick and easy". The message will be saved in string and using a loop, it will be displayed word by word.

```
import java.lang.Thread;
import java.lang.System;
import java.lang.Math;
import java.lang.InterruptedException;

class SiliconThread{
    public static void main(String args[]) {
    SMThread sm1 = new SMThread("Running Thread sm1
    : ");
    SMThread sm2 = new SMThread("Running Thread sm2
    : ");
    sm1.start();
    sm2.start();
    boolean Live_sm1 = true;
    boolean Live_sm2= true;
    do {
            if(Live_sm1 &&  !sm1.isAlive()){
            Live_sm1 =  false;
            System.out.println("1st thread is finished.");
            }
            if(Live_sm2 &&   !sm2.isAlive()){
            Live_sm2 = false;
            System.out.println("2nd thread is finished.");
```

```
            }
        }while(Live_sm1 || Live_sm2);
    }
    }

    class SMThread extends Thread {
            static String message[] =
            {"This","book","on,","Java,","is", "quick", "and",
            "easy"};
            public SMThread(String id) {
                super(id);
            }
            public void run() {
                String name = getName();
                for(int i=0;i<message.length;++i) {
                randomWait();
                System.out.println(name+message[i]);
                }
            }
            void randomWait(){
                try {
                sleep((long)(4000*Math.random()));
                }catch (InterruptedException x){
                System.out.println("Interrupted!");
                }
            }
        }
```

The above program creates two threads : sm1 and sm2, from the SMThread class which extends Thread class. It then starts both threads and executes a do ...while loop that stop execution of thread when it is complete. The threads display the "This book on Java is quick and easy" message word by word, while waiting a short, random amount of time between each word. Because threads are being processed on single CPU system, only one thread is executed at a time. This becomes obvious when the program's output identifies the threads that is being executed.

The output comes something like given below. You may get a different output as you run the program each time, the output will be different. This is because the random number that is being used to delay the display of each thread.

```
Running Thread sm2: This
Running Thread sm1: This
Running Thread sm1: book
Running Thread sm2: book
```

```
Running Thread sm1: on,
Running Thread sm2: on,
Running Thread sm1: java
Running Thread sm2: java
Running Thread sm1: is
Running Thread sm2: is
Running Thread sm1: quick
Running Thread sm2: quick
Running Thread sm1: and
Running Thread sm2: and
Running Thread sm2: easy
2nd thread is finished.
Running Thread sm1: easy
1st thread is finished.
```

The above output shows that thread sm2 is executed first and displayed "This" to the console window. It then waited to execute while thread sm1 displayed "This" and "book". Then thread sm1 waited while thread sm2 continued its execution and display "book". Now again thread sm2 waited and sm1 displayed "on". The process continues and the threads sm1 and sm2 are displayed till all the strings has been displayed. When execution is completed, message "1st thread is completed" and "2nd thread is completed" is displayed.

The **SiliconThread** class consists of a single **main()** method. This method begins by creating thread **sm1** and **sm2** as new objects of class **SMThread**. It then starts both threads using the **start()** method. At this point, **main()** enters a do loop that continues until both **sm1** and **sm2** are no longer alive. The loop monitors the execution of the two threads and displays a message when it has detected the death of each thread. It uses the **isAlive()** method of the Thread class to tell when a thread has finished.

The **SMThread** class extends class **Thread**. It declares a statically initialized array, named **message[]**, that contains the message to be displayed by each thread. It has a single constructor that invokes the Thread class constructor via **super()**. It also contains two access methods: **run()** and **randomWait()**.

The **run()** method uses the **getName()** method of class Thread to get the name of the currently executing thread. It then prints each word of the output display message while waiting a random length of time between each print. (4000*Math.random()).

The **randomWait()** method invokes the **sleep()** method within a try statement. The **sleep()** method is another method inherited from class Thread. It causes the currently executing task to "go to sleep" or wait until a randomly specified number of milliseconds has transpired. Also, the **sleep()** method throws the Interrupted Exception

when its sleep is interrupted, the exception is caught and handled by the **randomWait()** method.

IMPLEMENTING RUNNABLE

Here, you will create another program similar to previous program but the threads you create will be objects of a class (Say SMclass) which is not a subclass of Thread. Instead, the class SMclass will implement the Runnable interface and Objects of SMclass will be executed as threads by passing them as arguments to the Thread constructor.

```
import java.lang.Thread;
        import java.lang.System;
        import java.lang.Math;
        import java.lang.InterruptedException;
        import java.lang.Runnable;

        class SiliconThread1{
            public static void main(String args[]) {
            Thread sm1 = new Thread(new SMclass("Running
            Thread sm1: "));
            Thread sm2  = new Thread(new SMclass("Running
            Threadsm2 :"));
            sm1.start();
            sm2.start();
            boolean Live_sm1 = true;
            boolean Live_sm2= true;
            do {
                    if(Live_sm1 &&  !sm1.isAlive()){
                    Live_sm1 =  false;
                    System.out.println("1st thread is finished.");
                    }
                    if(Live_sm2 &&   !sm2.isAlive()){
                    Live_sm2 = false;
                    System.out.println("2nd thread is finished.");
                    }
            } while(Live_sm1 || Live_sm2);
        }
        }

        class SMclass implements Runnable{
                    static String message[] =
                    {"This","book","on,","Java,","is", "quick", "and",
                    "easy"};
                    String name;
```

```
        public SMClass(String id) {
                name = id;
        }
        public void run() {
        //      String name = getName();
                for(int i=0;i<message.length;++i) {
                randomWait();
                System.out.println(name+message[i]);
                }
        }
        void randomWait(){
                try {
                Thread.currentThread().sleep
                (long)(3000*Math.random()));
                } catch (InterruptedException x){
                System.out.println("Interrupted!");
                }
        }
}
```

You may note the difference between this method and the previous method. The changes in this method are underlined in the above program. If you run the program, the output is similar to what was shown by earlier program.

Here, the **main()** method differs in the way that it creates thread **sm1** and **sm2**. The program first created instances of **SMclass** and then passed them to the **Thread()** constructor, creating instances of class Thread. The **Thread()** constructor takes as its argument any class that implements the Runnable interface.

SMclass is declared as implementing the Runnable interface. This is a simple interface to implement; it only requires that you implement the **run()** method. **SMclass** declares the name variable to hold the names of **SMclass** objects that are created.

The **run()** methods is identical to the previous program, except with the name issue. This is also true of the **randomWait()** method. Here, the **randomWait()** method must use the **currentThread()** method of class Thread to acquire a reference to an instance of the current thread in order to invoke its **sleep()** method.

The advantage of using the Runnable interface is that your class does not need to extend the Thread class. This will be very helpful feature when you start using multithreading in applets in the coming chapters. The only disadvantages to this approach are ones of convenience. You have to do a little more work to create your threads and to access their methods.

THE THREAD API

Constructor

The Thread class has seven different constructors :

public Thread();
public Thread(Runnable target);
public Thread(Runnable target, String name);
public Thread(String name);
public Thread(ThreadGroup group, Runnable target);
public Thread(ThreadGroup group, Runnable target, String name);
public Thread(ThreadGroup group, String name);

These constructors has three different parameters - thread name, thread group, and a Runnable target object.

- name is the (string) name to be assigned to the thread. Every java thread must have a name. You can set the name during construction or with setname() property.

  public final void setName(String name);

  If you fail to specify a name, the system generates a unique name of the form Thread-N, where N is a unique integer.

  You can retrieve the name of a thread using the getName() method.

  public final String getName();

  Although Java assign the thread names but still it is good practice to assign the names as they provide the programmer with a useful way to identify particular threads during debugging.
- target is the Runnable instance whose run() method is executed as the main method of the thread.
- group is the ThreadGroup to which this thread will be added.

Starting and Stopping

To start and stop threads once you have created them, you need the following methods:

- To start a new thread, you must call the thread by start() method. An exception is thrown if start() is called more than once on the same thread.

  public void start();
- There are two main ways a thread can terminate: The thread can return from its run() method, ending gracefully. Or the thread can be terminated by the stop() or destroy() method.

  public final void stop();
  public final void stop(Throwable obj);
  public void destroy();

- The stop() method causes that thread to terminate by throwing an exception to the thread (a ThreadDeath exception). When you stop a running thread, it might be doing something at that time and will be using some system resources. If you stop the thread at such moment, it will cause all activity on the thread to cease immediately and these resources might not be cleaned up properly. If you stop a thread at the wrong moment, it would be unable to free these resources which may lead to potential problems for the virtual machine.
- To provide for clean thread shutdown, the thread to be stopped is given an opportunity to clean up its resources. A ThreadDeath exception is thrown to the thread, which percolates up the thread's stack and through the exception handlers that are currently on the stack.
- The destroy() method is stronger than the stop() method. The destroy() method is designed to terminate the thread without resorting to the ThreadDeath mechanism. The destroy() method stops the thread immediately, without cleanup; any resources held by the thread are not released.

THREAD PRIORITY AND SCHEDULING

We have seen that in a single processor system, the threads do not execute at the same time. Instead, the threads share execution time with each other based on the availability of the system's CPU (or CPUs). However, in multiprocessor system, the threads may execute at same time.

Now, when program execute on a single processor system, it may execute any thread, as shown in Fig 4.1. However, using scheduling mechanism, you can control the execution of threads. Basically, scheduling determines how the threads in ready state should be allocated the CPU time?

Scheduling is performed by the Java runtime system. As Java is intended to be platform independent, it schedules threads based on their priority. Irrespective of the scheduling policies of operating system or thread packages, the highest-priority thread, that is in the runnable state, is the thread that is run at any given instant. The highest-priority thread continues to run until it enters the death state, enters the not runnable state, or has its priority lowered, or when a higher-priority thread becomes runnable.

This approach of scheduling in Java, is referred to as preemptive scheduling. When a thread of higher priority becomes runnable, it preempts threads of lower priority and is immediately executed in their place. If two or more higher-priority threads become runnable, the Java scheduler alternates between them when allocating execution time.

Setting Thread Priority

Every thread has a priority. The thread inherits the priority from the thread from which it is created. You can also set the priority using setPriority() method. You can also get the priority using getPriority(). Following is the setting of Priority in Java API:

```
public final static int MAX_PRIORITY = 10;
public final static int MIN_PRIORITY = 1;
public final static int NORM_PRIORITY = 5;
public final int getPriority();
public final void setPriority(int newPriority);
```

In Java API, thread priority is expressed as an integer value ranging from 1 to 10, 10 being the highest. Thread class defines variables MIN_PRIORITY, NORM_PRIORITY and MAX_PRIORITY that have values 1, 5, and 10 respectively. A thread has NORM_PRIORITY by default.

Waking Up a Thread

Following methods relating to thread wakeup are declared in Java API:

```
public void interrupt();
public static boolean interrupted();
public boolean isInterrupted();
```

You can use interrupt() method on the thread object to send a wake-up message. It causes an InterruptedException to be thrown in the thread. It also sets a flag that can be checked by the running thread using the interrupted() or isInterrupted() method.

Thread.interrupted() checks the interrupt status of the thread and sets the interrupt status to false, whereas Thread.isInterrupted() only checks the interrupt status of the thread but doesn't change the status.

Suspending and Resuming Thread

To suspend a thread execution, we have suspend() method.

```
public final void suspend();
```

When you call the suspend() method, it ensures that a thread will not be run. To reverse the suspend() method, you have resume() method.

```
public final void resume();
```

However, calling resume() method doesn't always ensure that the target thread will become runnable - other events may have caused the thread to be not runnable.

Sleep() method

You can use sleep() method to pause the current thread for a specified period of time. There are two methods that has been defined for sleep().

public static void sleep(long millisecond);
public static void sleep(long millisecond, int nanosecond);

E.g. Thread.sleep(500) will pause the current thread for half a second. During this time the thread would not be in runnable state. When the specified time expires, the current thread again becomes RUNNABLE.

Waiting for a Thread to End

If you want to wait for a specific thread to end before you could start a new task, is referred to as joining the thread and you can take help of one of the join() methods.

public final void join();
public final void join(long millisecond);
public final void join(long millisecond, int nanosecond);

```
E.g.    SMThread sm1 = new SMThread("Thread sm1: ");
        sm1.start();
        ..............
        sm1.join();
```

The join() method with no parameters, waits for the thread to terminate. However, if you specify a time parameter with join() method, the join() method returns after specified time, irrespective of the fact whether the thread has terminated or not.

Daemon Threads

When you require threads which run in the background to provide services to other threads, are called Daemon threads.

public final boolean isDaemon();
public final void setDaemon(boolean on);

The daemon thread typically executes a continuous loop of instructions that wait for a service request, perform the service, and wait for the next service request.

The Java virtual machine (VM) has a default daemon thread, known as the garbage collection thread. It is a low priority thread, executing only when there is nothing else for the system to do.

You can use the **setDaemon()** method to set the daemon status of thread. To check whether a thread is a deamon thread or not, you may use **isDaemon()** method. This method returns true if this thread is a daemon thread.

Other Thread Methods

- To return the number of active stack frames (method activations) currently on this thread's stack, use countStackFrames().

  public int countStackFrames();

- To return the ThreadGroup class to which this thread belongs, you can use the getThreadGroup() method..

  public final ThreadGroup getThreadGroup();

- Use **isAlive()** method to find out whether the thread is dead on not. It returns true value, if this thread is RUNNABLE or NOT RUNNABLE and false if this thread is NEW or DEAD.

  public final boolean isAlive();

- To return the Thread object for the current sequence of execution use **currentThread()** method.

  public static Thread currentThread();

- To return the number of threads in the currently executing thread's ThreadGroup class, you can use activeCount() method.

  public static int activeCount();

- enumerate(Thread array[]) method returns a list of all threads in the current thread's ThreadGroup class.

  public static int enumerate(Thread tarray[]);

- To prints a method-by-method list of the stack trace for the current thread to the System.err output stream, you can use dumpStack() method..

  public static void dumpStack();

- toString() method returns a debugging string that describes this thread.

  public String toString();

THREAD GROUPS

Thread groups are objects that consist of a collection of threads. Every thread belongs to only one thread group. The Threadgroups class helps to organize and manage of similar groups of threads.

Every thread becomes a member of the thread group as soon as it is created. It remains member of the same group throughout its existence. A thread can never become a member of another group.

THREADGROUP API

Constructors

The ThreadGroup class has following two constructors:

public ThreadGroup(String name);

public ThreadGroup(ThreadGroup parent, String name);

Both of these constructors require you to specify the new thread group. The first constructor which doesn't take the parent name, creates the new group as a child of the currently executing thread group.

Priority

You can also manage the priority of the threads inside a group using setMaxpriority() method. When this method is called, no thread within the group can use setPriority() method to set a priority higher than the specified maximum value.

public final int getMaxPriority();

public final void setMaxPriority(int pri);

ThreadGroup Tree

- Thread Group is like a folder. It could contain other Thread Group and Threads. There are two methods which gives information about the threads and groups.

  public int activeCount();

  public int activeGroupCount();

  To count the number of threads that are member of ThreadGroup tree, you can use **activeCount()** method. To count the number of ThreadGroups that are members of any ThreadGroup, use **activeCountGroup()** method.

- To list the threads or group in the ThreadGroup object, you can use one of the following enumerate() methods :

  public int enumerate(Thread list[]);

  public int enumerate(Thread list[], boolean recurse);

  public int enumerate(ThreadGroup list[]);

  public int enumerate(ThreadGroup list[], boolean recurse);

  The **enumerate()** method with recurse parameter having true value, list all the threads and groups inside the ThreadGroup. If recurse is false, only the threads or groups in this immediate ThreadGroup object are retrieved. If no recurse parameter is used with enumerate() method, it is similar to the situation, where recurse parameter is true.

- You can use the **parentOf()** method to check whether the current thread group is a parent of specified group. It returns true if this thread group is the parent of the specified group otherwise false.false otherwise. Following is this method's syntax:

  public final boolean parentOf(ThreadGroup g);

- If you want to find out the parent of the thread group, you can use the **getParent()** method. It returns the parent of this thread group, or null if this ThreadGroup is the top-level ThreadGroup. Following is this method's syntax:

  public final ThreadGroup getParent();

Other ThreadGroup Methods

- To return the name of the threadgroup, you can use the **getName()** method. Following is this method's syntax:

  public final String getName();

- Similar to threads, the ThreadGroups can also be referred as Daemons. When a ThreadGroup object is a daemon group, the group is destroyed, once all its threads and groups have been removed.

  public final boolean isDaemon();
  public final void setDaemon(boolean daemon);

  To find out whether a ThreadGroup is Daemon group, you can use **isDaemon()** method.

- To display the debugging information about the thread group, you can use **toString()** method.

  public String toString();

SYNCHRONIZATION

We discussed in the beginning of the chapter that the main advantage of Java is that it can run multiple threads. Running multiple threads is natural requirement from user point of view. e.g. simultaneously downloading a file from the Internet, performing a spreadsheet recalculation, and printing a document. This is called concurrency. Programmers have to take extra care to design programs which can run multiple threads.

The real problem lies when two or more threads are accessing the same object. For example, when one thread is updating the data and second thread is reading the same data. Concurrency requires the programmer to take special precautions to ensure that Java objects are accessed in a thread-safe manner. Over the years, many concurrency-control solutions have been proposed and implemented.

These solutions include Critical sections, Semaphores, Mutexes, Database record locking and Monitors. Java implements a variant of the monitor approach, introduced by C. A. R. Hoare in a 1974, to concurrency.

Monitors in Java enforce mutually exclusive access to synchronized methods. You coordinate the actions of thread using synchronization method and synchronized statements. Any method, in Java, is referred as synchronised method, if it has a synchronised keyword before the return type. Only one synchronized method can be invoked for an object at a given point in time. This keeps synchronized methods in multiple threads from conflicting with each other.

Every class and object is associated with a unique monitor. The monitor is used to control the way in which synchronized methods are allowed to access the class or object. When a synchronized method is invoked for a given object, it acquires the monitor for that object. No other synchronized method may be invoked for that object until the monitor is released. A monitor is automatically released when the method completes its execution and returns.

Let us consider the example discussed in the beginning of chapter. Here, we are trying to control the running of the two threads so that the second thread starts, when first thread is finished.

```
import java.lang.Thread;
import java.lang.System;
import java.lang.Math;
import java.lang.InterruptedException;

class SiliconThread{
    public static void main(String args[]) {
    SMThread sm1 = new SMThread("Running Thread sm1
    : ");
    SMThread sm2 = new SMThread("Running Thread sm2:
    ");
    sm1.start();
    sm2.start();
    boolean Live_sm1 = true;
    boolean Live_sm2= true;
    do {
            if(Live_sm1 &&  !sm1.isAlive()){
            Live_sm1 =  false;
            System.out.println("1st thread is finished.");
            }
            if(Live_sm2 &&   !sm2.isAlive()){
            Live_sm2 = false;
            System.out.println("2nd thread is finished.");
            }
    }while(Live_sm1 || Live_sm2);
}
}

class SMThread extends Thread {
            static String message[] =
            {"This","book","on,","Java,","is", "quick", ”and”,
            ”easy”};
            public SMThread(String id) {
                    super(id);
            }
            public void run() {
            SyncOutput. display(getName(), message);
            }
            void randomWait(){
                    try {
                    sleep((long)(4000*Math.random()));
```

```
                    }catch (InterruptedException x){
                    System.out.println("Interrupted!");
                    }
                }
            }
        class SyncOutput {
            public static synchronized void display(String
            name,String list[]) {
            for(int i=0;i<list.length;++i) {
            SMThread sm = (SMThread) Thread.currentThread();
            sm.randomWait();
            System.out.println(name+list[i]);
            }
        }
        }
```

The output of the above program is as given below:

```
Running Thread sm1: This
Running Thread sm1: book
Running Thread sm1: on,
Running Thread sm1: Java,
Running Thread sm1: is
Running Thread sm1: quick
Running Thread sm1: and
Running Thread sm1: easy
1st thread is finished.
Running Thread sm2 :This
Running Thread sm2 :book
Running Thread sm2 :on,
Running Thread sm2 :Java,
Running Thread sm2 :is
Running Thread sm2 :quick
Running Thread sm2 :and
Running Thread sm2 :easy
2nd thread is finished.
```

The above program is similar to the one we ran in the beginning of chapter. The only difference is that run() method has been changed. Earlier the output was coming directly from run() method. Here, run() method invokes the display() method of the class SyncOutput.

This has been done to incorporate the synchronization. As we know that we can synchronize a method, therefore we introduced a method display() in the class SyncOutput, which has the keyword synchronized.

When you run the program, thread sm1 invokes display() method, acquires a monitor for the SyncOutput class and display() method proceeds with the display of thread sm1. Because thread sm1 acquired a monitor for the SyncOutput class, no other thread can acquire this class till the present thread sm1 is finished. Therefore, thread sm2 must wait until the monitor is released before it is able to invoke display() to display its output.

If you remove the keyword synchronized, and run the program, the output will be unsynchronized, i.e. the threads will be executed randomly.

INTER THREAD COMMUNICATION

From above, it is clear that Monitors act as object lock. Monitors can also be used to coordinate multiple threads by using the wait() and notify() methods available in every Java object.

The synchronized keyword blocks out the other threads unconditionally from access to that method. At times, it is required to give control to the other threads from within the synchronized method. In such cases, wait() and notify() methods are used.

The wait() method unlocks the monitor from the current thread and put the thread to sleep until some other thread enters the same monitor and calls notify() method. The notify() method will wake up the first thread that called *wait()* on the same object. In case there are many threads that have been suspended using wait() method on the same object, you can resume all of them by using notifyAll() method.

There are two additional varieties of the wait() method. The first version takes a single parameter - a timeout value in milliseconds. The second version has two parameters - a more precise timeout value, specified in milliseconds and nanoseconds.

wait(long milliseconds);
wait(long milliseconds, int nanoseconds);

EXERCISE

1. Fill in the blanks:
    1. A thread enters a _____ state as soon as it is created.
    2. In multithreading a program is broken into _____ and then these _____run in parallel.
    3. The start() method of the Thread class automatically invokes the _____ method.
    4. The priority of a thread is an integer value ranging from _____ to _____.
    5. Threads can be created in two ways _____ and _____.
    6. The default priority of a thread is _____.
    7. Java uses a concept called _____for inter-process synchronization.
    8. _____ method is used to suspend a thread for some time.
    9. _____ method gets the name of a thread.
    10. _____ method wakes up all the threads that have been suspended using wait() method.
2. What are Java Threads? What are the various states of Java threads?
3. How the threads are created? Discuss the Subclass and Runnable method and their advantages and disadvantages.
4. What do you understand by sleeping and waking up of thread?
5. What are Daemon Threads?
6. What is thread group? Discuss the API of thread group in brief?
7. What is thread synchronization?

EXCEPTION HANDLING

EXCEPTIONS
HANDLING EXCEPTION
CUSTOMIZED EXCEPTIONS

Exception Handling

Errors are normal part of programming. The development of reliable, error-tolerant software is a multiphase effort that spans program design, coding, compilation, loading, and execution. When you develop some program, you anticipate major areas where error could occur and provide remedy for that. But even then, you are unable to anticipate many error and these errors are located by running the program thousands time under different test conditions. (This is the reason why beta version of every software is launched before bringing the final version).

The Java language eliminates whole classes of errors that result from the use of dangerous programming constructs. The more simple and familiar is the language, the less are the chances of occurrence of programming errors.

EXCEPTIONS

An exception is exceptional condition in programming. When such a condition arise, Java gives it own message regarding the exception. However, the message may not be so much user friendly. To make the message more user friendly, you can write a code which is executed when exceptional condition arise. Thus, Exceptions provide notification of errors and a way to handle them. By using the exception handling you can prevent your program from abruptly ending, flashing surprising and annoying error messages. On the contrary, if a minor error crops up it can be handled during runtime and your program resumes from there on without crashing down.

Java provides superior support for runtime error and exception handling, allowing programs to check for anomalous conditions and respond to them with minimal impact on the normal flow of program execution. This allows error - and exception-handling code to be added easily to existing methods.

Exceptions are generated by the Java runtime system in response to errors that are detected when classes are loaded and their methods are executed. The runtime system is said to throw these runtime exceptions. Runtime exceptions are objects of the class java.lang.RuntimeException or of its subclasses.

Exceptions may also be thrown directly by Java code using the throw statement. These exceptions are thrown when code detects a condition that could potentially lead to a program malfunction. The exceptions thrown by user programs are generally not objects of a subclass of RuntimeException. These non-runtime exceptions are referred to as program exceptions.

HANDLING EXCEPTION

Both program and runtime exceptions must be caught in order for them to be processed by exception-handling code. If a thrown exception is not caught, its thread of execution is terminated and an error message is displayed on the Java console window.

The approach used by Java to catch and handle exceptions is to surround blocks of statements for which exception processing is to be performed with a try statement. The try statement contains a catch clause that identifies what processing is to be performed for different types of exceptions. When an exception occurs, the Java runtime system matches the exception to the appropriate catch clause. The catch clause then handles the exception in an appropriate manner.

TRY AND CATCH

The exception thrown by runtime system are caught in order for them to be processed by exception-handling code. To respond to the exception, you must place the call to the method that produces the exception within a try block.

A try block is a block of code beginning with the try keyword followed by a left and right curly brace. Every try block is associated with one or more catch blocks. Here is a try block:

```
try
   {
       // method calls
   }
```

When you place the call in the try block, method will catch the exception thrown by the method it calls. To handle the exception thrown, you can have the catch block. Different catch blocks handle different types of exceptions.

Consider the following code, which has a try block and catch block to handle exception of type Exception.

```
try
{
    // method calls
}
catch( Exception x )
{
    // handle exceptions
}
```

Inside the try block, as soon as the exception condition arises, an object of that specific type of exception is automatically created and

thrown. As soon as the try block throws any type of exception, execution of the try block ceases and control is transferred to catch block. The thrown object is then matched with the catch block. If the block matches, then the statements for handling the exception written within the catch block are executed.

If the block doesn't matches, the exception is passed to the method's caller. Again the exception is compared with the catch block. The process continues until a catch block catches the exception. If no catch block is able to catch the exception, it reaches to the main() method uncaught and causes the application to cease.

Consider the example given below:

```
class ArthExcp
    {
    public static void main(String args[ ])
    {
    try
            {
            int count1 = 6;
            int count2 = 12;
            int k = count1/(count2 - 2* count1);
            System.out.println(k);
            }catch(ArithmeticException e)
            {
            System.out.println("##Wrong Count 2##");
    }
    }
}
```

The above program, when compiled and gives the output as

```
##Wrong Count 2##
```

The catch block catches the Arithmetic exception and accordingly the output is displayed. If you don't define the Arithmetic Exception (Division by zero is ArithmaticException) in catch statement and define some other exception - say NumberFormatException, the following message will be displayed:

```
Exception in thread "main" java.lang.ArithmaticException
: / by zero
at ArthExcp.main(ArthExcp.java : 9)
```

The above message, is the default error, if appropriate exception is not defined in catch block.

THROW

In the above example, the exception has been thrown automatically. Such exceptions are called implicit exceptions. At times it is required

to throw the exception explicitly. This is used when you create your own exceptions and you want to throw them. All the system-defined exceptions are thrown automatically, but the user-defined exceptions must be thrown explicitly using the throw clause. It takes the form as:

```
try
{
    // statements
    throw new Udef_Exception();
    //statements
    }catch(Udef_Exception obj1)
    {
    System.out.println("User defined exception caught");
}
```

Here, the exception is thrown using throw and catch block catches the exception. The Udef_Exception is a class made specifically to handle an exception and it is encapsulating some specific kind of user defined behaviour.

Consider the following set of instructions, where, exception is thrown:

```
class Throw
{
    public static void main(String args[ ])
    {
    int a = 5;
    int b = 10;
    try
    {
            if (b > a)
            throw new NullPointerException();
            }catch(NullPointerException e)
            {
            System.out.println(" B is greater than A");
            }
    }
}
```

We have deliberately thrown an exception, when a particular condition is satisfied. As soon as the exception is raised, it is caught with the catch clause, the statements inside the catch clause are executed and we get the output as:

```
B is greater than A
```

which shows that exception was thrown and catch block was executed.

THROWS

Till now, you have seen that the exception was thrown in the try block and caught by the catch block in the same method. At times you may require a method which throws an exception, but doesn't catch it in the method body. Rather you want to catch the exception in other method body.

In such a case, the method which is throwing the exception uses a throws clause with its definition. The throws clause acts as warning message to the calling module that the called module is throwing an exception which it is not handling, and it has to handle the thrown exception.

```
public static void main(String args[ ])
    {
            try
            {
                    method_1();
            }catch(<exception_1>  object)
            {
                    ..............
            }
            }
    }
class <classname>
    {
            static void method_1() throws <exception_1>
            {
            throw new <exception_1>
            }
```

Here main method calls method_1, but is not throwing any exception. method_1 throws an exception exception_1 but doesn't catch it. The exception moves to the caller of method_1 - i.e. main() which catches it and displays the required output.

Let us consider a case where main() calls a method publisher() and publisher() calls another method book(). the book() method throws an exception but doesn't has a provision to catch it. The catching opportunity moves to publisher() method, which can catch the exception if it has the catch block. If publisher() also doesn't have the catching facility, the exception can be caught by publisher() caller, main().

Here, both publisher() and book() do not have the catch block. Therefore, they must send the warning using **throws** keyword as shown below:

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

```
import java.lang.Exception ;
public class Throws {
    public static void main( String args[] ) {
    try
    {
            publisher() ;
    }catch( Exception e )
    {
    System.out.println( "Caught publisher() and Book()
    without Catch block") ;
    }
    }
static void publisher() throws Exception {
            book() ;
    }
static void book() throws Exception {
            throw new Exception() ;
    }
}
```

The output of the above set of instructions is :

```
Caught publisher() and Book() without Catch block
```

MULTIPLE CATCH

The try block may have a list of things to do and in the process, it could throw more than one expectation. It is similar to playing shots in different direction on the cricket field. Therefore, we need a fielder at various positions, if we want to catch the shot. Same principle applies here, you must provide enough catch block (our fielder) to catch all the exceptions thrown by the try block.

When an exception is thrown it traverses through the catch blocks one by one until a matching catch block is found. The program structure in such a case is:

```
try {
    ..........
}
catch(<exception_one> obj){
    ..........
}
catch(<exception_two> obj){
    ..........
}
catch(<exception_three> obj){
    ..........
```

```
}
```

Consider the following example, where various exceptions have been defined for the try block.

```
class ArthExcp1
    {
    public static void main(String args[ ])
    {
    String message[] = {"This","code","has","more", "than",
    "defined", "strings"};
    try
            {
            for(int ctr = 0; ctr <=message.length; ctr ++){
            System.out.println(message[ctr]);
            int count2 = 12;
            int k = count2/ctr;
            System.out.println(k);
            }
            }
            catch(ArrayIndexOutOfBoundsException  e){
            System.out.println("More arrays than defined");
            }
            catch(ArithmeticException e){
            System.out.println("##Wrong ctr##");
            }
    }
    }
```

The above program is expected to throw more than one expectation. Therefore two catch block are defined. Now you compile and run the program. The try block has a for loop with counter **ctr** which is being used in arithmetic expression. The initial value of ctr is 0, therefore just after printing the first String "This", the program gives the message ##Wrong ctr##. The output is like this:

```
This
##Wrong ctr##
```

The exception was ArithmeticException as you were trying to divide by zero. Thus second catch block was executed.

Now, change the for statement as follows:

```
for(int ctr = 1; ctr <=message.length; ctr ++)
```

i.e. start the ctr from 1 and run the program. You get the following output:

```
code                    message[1]
12                      k = 12/1 = 12
```

has	message[2]
6	k = 12/2 = 6
more	message[3]
4	k = 12/3 = 4
than	message[4]
3	k = 12/4 = 3
defined	message[5]
2	k = 12/5 = 2
strings	message[6]
2	k = 12/6 = 2
More arrays than defined	**Exception**

Now you can see that the exception ArrayIndexOutOfBoundsException was executed as ctr has the expression ctr <= message.length. Thus the loop tries to display String message[7] , which is out of Upper bound. Thus the first catch block is executed.

FINALLY

The finally clause is another clause that can appear with try block. It can either substitute the catch block or appear along with the catch block. The finally clause ensures that the block of the clause must be executed after the try block, irrespective of the fact whether the exception was thrown by the try block or not. It could take one of the following form:

```
try{
    ..............
    }finally {
    ..............
    }
```

or

```
try{
    ..........
    }catch(<exception> obj){
    ............
    }
    finally{
    ..............
    }
```

Consider the following example, which throws an ArithmeticException and then execute the finally block.

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

```
        {
        try
                {
                int count1 = 6;
                int count2 = 12;
                int k = count1/(count2 - 2* count1);
                System.out.println(k);
                }catch(ArithmeticException e)
                {
                System.out.println("##Wrong Count 2##");
                }
                finally
                (
                System.out.println("Executing Finally");
                }
        }
}
```

The output of the above program comes as follows:

```
##Wrong Count 2##
Executing Finally
```

Thus, the finally block is always executed, irrespective of the fact that whether try block throws an exception or not.

NESTED TRY BLOCK

You can use nested try statements to provide multiple levels of exception-handling capabilities. This is accomplished by enclosing a method or block of statements containing a lower-level try statement within the try block of a higher-level try statement.

```
try                                    // Higher level
    {
    .........
            try                        // Lower Level
            {
            ..................
            }catch(<exception_ lower Level>){
            ............
            }
    }catch(<exception_higher Level> ){
            ............
    }
```

When an exception is thrown in the try block of the lower-level try statement that cannot be caught, it continues to be thrown until it reaches the higher-level try statement. The higher-level try statement

can then determine whether the exception can be caught and processed by any of its catch clauses. Any number of try statements can be nested.

Consider the following example, where nested try blocks have been used.

```
class ArthExcp1
        {
        public static void main(String args[ ])
        {
        String message[] = {"This","code","has","more", "than",
        "defined", "strings"};
        try
        {
        for(int ctr = 0; ctr <=message.length; ctr ++){
        System.out.println(message[ctr]);
                try{
                        int count2 = 12;
                        int k = count2/ctr;
                        System.out.println(k);
                }catch(ArrayIndexOutOfBoundsException  e){
                System.out.println("More arrays than defined");
                }
                }
        }catch(ArithmeticException e){
        System.out.println("##Wrong ctr##");
        }
        }
        }
```

Here, outer try statement is displaying the strings, whereas inner try block is calculating the integer k. The inner try block takes the exception ArrayIndexOutOfBoundsException, while outer try block takes the ArithmeticException.

When you execute the above code, you get the following output

```
This
##Wrong ctr##
```

You may note that the message that appears is for ArithmeticException while Arithmetic exception is thrown by the inner try block. Thus the exception traverse from inner blocks to outer blocks.

CUSTOMIZED EXCEPTIONS

Java provides many built-in classes for exception handling, which has been defined in Appendix A. Besides these built-in classes, it also

provides the facility of defining your own exception classes. This is very useful when you want to define exception types which are having a behaviour different from the standard exception types, particularly when you want to do validations in your application.

Consider the following example, which creates a user-defined exception and uses it.

```
class New_Excp extends Exception
{
    String message;
    New_Excp()
    {
    message = new String("This is User Defined Exception");
    }
    public String toString()
    {
            return (message);
    }
    }
public class My_Exception
{
    public static void main(String args[ ])
{
try{
    throw new New_Excp();
    }catch(New_Excp e)
    {
            System.out.println(" UDE : " + e);
    }
}
}
```

The output of the above java program is :

UDE : This is User Defined Exception

The user defined class New_Excp is derieved from Exception class. For any class to behave as an exception type it must become the subclass of the Exception class.

The class defines a variable **message**, and intializes it with appropriate message.

EXERCISE

1. What are exceptions? Differentiate between an exception and error.
2. Describe the structure of the exception handling block.
3. Explain the significance of following and write the syntax?
   a) Try and Catch
   b) Throw
   c) Throws
   d) Multiple Catch
4. Explain the significance of the throws clause.
5. Do you use nested try statements just to introduce more clarity in your program? If not then why?
6. State True or False?
   a) Every try block is associated with exactly one catch blocks.
   b) Exceptions are always thrown automatically.
   c) The finally block does not execute if a catch block executes
   d) The throw clause uses the exception name with it.
   e) The finally clause ensures that the block of the clause must be executed after the try block.
   f) In nested try block, exception is thrown until it reaches the higher-level try statement.

JAVA APPLETS

APPLET LIFE CYCLE

VIEWING APPLET

Java Applets

Java is the most favourite language of Web developers all over the world. This one fact played an important role in rising the java popularity to such a height. Java provides the concept of applets which is a Java program that can operate only within a compatible Web browser, such as Netscape Navigator or Microsoft Internet Explorer or Sun's HotJava. When a Java applet is encountered on a page, it is downloaded to the user's computer and begins running.

Programming applets with Java is different from creating applications with Java. As applets are downloaded off a page each time they are run, applets are smaller than most applications to reduce download time. Also, because applets run on the computer of the person using the applet, they have numerous security restrictions in place to prevent malicious or damaging code from being run.

In early days of development, Java applets were mostly used for animating graphics. Today, Java applets are used to accomplish far more demonstrative goals such as Tickertape-style news and sports headline updates, Video games, Student tests and quiz, Interactive Geographical map, Advanced text displays, Database reports and may more.

Java also provides the windows and other graphical user interface are also provided and handled by Abstract Windows Toolkit (java.awt package). The AWT package lets you create all type of graphical user interface including Windows and dialog box, pull down menus, buttons, labels check box and all other user interface components. It also provides event handling to respond to mouse clicks, mouse movements, and keyboard input. It will be discussed in more details in the coming chapters.

Also, there are many tools available for developing java applications and applets. It includes, but not limited to, SunSoft Java WorkShop, Symantec Café and Visual Café, and Rogue Wave JFactory. These tools offers visual programming techniques for developing applications and applets using Java.

SECURITY IN APPLETS

As you know, any thing that can execute code is potential security risk. The risk becomes more when codes are being downloaded from web. Security is one of the primary concerns of Java's developers, keeping that in mind, some of Java's functionality is blocked for applets. In general, the applets are different from application in following aspect :

- The applet execution depends on the browser setting. If the browser has been set for high security restrictions, the applet cannot read, write or execute files on that browser.

- Applets can communicate only with the site from where the applet was run.
- Applets cannot load programs like executable files and shared libraries from the machine where browser is displaying the applet.
- An applet requires a browser to run it. Therefore, it takes longer to execute an applet than an application. The amount of memory that can be used by an applet is decided by the browser, whereas there is no such restriction on applications.

APPLET LIFE CYCLE

You have already seen, how we can create and run an applet using browser, in chapter 1. Applet always requires a browser around it to be able to get executed. No applet can run outside browser. Unlike application, it doesn't have a main() method which is the starting point of application. Rather each applet starts out with a class definition, like this:

```
public class First_Applet extends java.applet.Applet {
        ..............
}
```

First_Applet is the name of the applet class which is always declared as a public class. All applets are derived from java.applet.Applet class, also referred as the Applet class.

```
import java.applet.Applet
public class First_Applet extends Applet {
    ..............
}
```

The java.applet package is the smallest package in the Java API. It consists of a single class - the Applet class - and three interfaces: AppletContext, AppletStub, and AudioClip.

The Applet class contains a single default parameterless constructor, which is generally not used. Applets are constructed by the runtime environment when they are loaded and do not have to be explicitly constructed.

The Applet class contains 21 access methods that are used to display images, play audio files, respond to events, and obtain information about an applet's execution environment, referred to as the applet's context.

These 21 methods also contains five methods - init(), start(), stop(), destroy() and paint() - that take place during the life of an applet. When each stage is reached, a method is automatically called.

init() The program is loaded for the first time

paint()	Something happens that requires the applet window to be redisplayed
stop()	The program stops at a specific point
start()	The program restarts after a stop
destroy()	The program is unloaded as it finishes running

init();

Initialization of any methods and variables takes place at this stage. The init method is called when the applet is first loaded. Thus you can use the init() method to set the initial behaviour of applet. You can set fonts, load images, initialize variables or set parameters here. To give an applet the type of functionality you want, you must override the init() method. The init() method takes the following form :

```
public void init(){
                .....................}
```

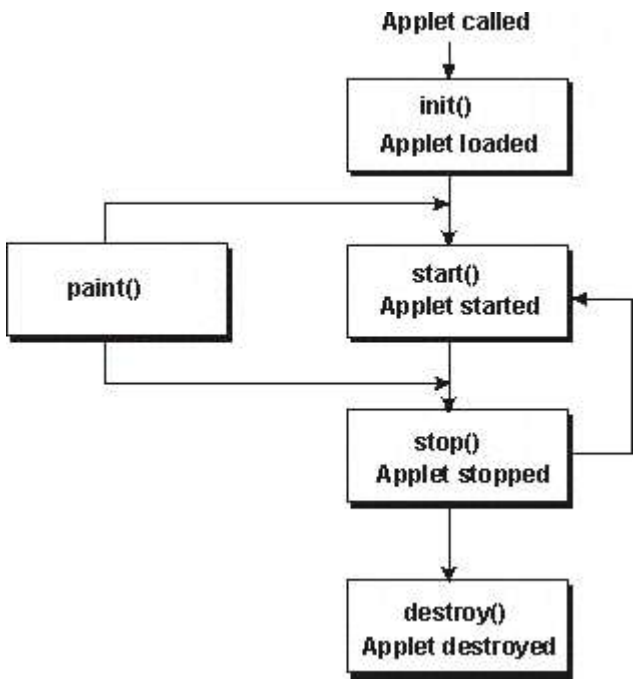


Fig 6.1 Applet Flow Chart

start();

This stage starts the primary functions of an applet. For example, if an applet plays a sound, it could start playing during this stage. This method is called when the init method has finished and the applet is ready to execute. This method is also called whenever user comes back to the page that contained the applet in the browser. The main difference between init() and start() is that while init() is called only once, start() can be called any number of times.

```
public void start(){
        .....
        }
```

stop();

This stage is used to stop any actions that is still in progress from the start stage when an applet is exited. For example, a sound loop would need to be stopped when a person left the Web page containing a sound applet. This method is called when the page is left, or the browser is minimized. You should always stop an applet before you destroy it; you can also use the stop() method to stop applet execution when a pause in the flow is needed.

```
public void stop(){
        .....
        }
```

destroy();

Destroy is called automatically, and completes any memory cleanup or garbage collection that needs to be done when an applet is destroyed. This method is called when you quit a browser. This method is the last thing that happens when the user leaves the page containing the applet.

```
public void destroy(){
        .....
        }
```

paint();

It is one of the most widely used method in any applet. Whenever something needs to be displayed or redisplayed on the applet window, the paint() method handles the task.

These methods also occur automatically at certain times, such as when the applet window is redisplayed after being covered or when the applet window is resized. This function accepts an object of the type Graphics, therefore you must import Graphics class of awt package in the applet.

```
public void paint(Graphics g){
        .....
}
```

You have to add the following import statement before the class statement at the beginning of the source file, as you will be using a Graphics object in your applet:

```
import java.awt.Graphics;
```

You also can force paint() to be handled with the following statement:

```
repaint();
```

An applet can call repaint() directly to update the window whenever necessary.

VIEWING APPLET

For viewing applets, you must have the basic knowledge of HTML and HTML tags. Applets are placed on a Web page in the same way that anything is put on a page. HTML commands are used to describe the applet, and the Web browser loads it along with the other parts of the page.

If you have used HTML to create a Web page, you know that it's a way to combine formatted text, images, sound, and other elements together. HTML uses special commands called tags that are surrounded by < and > marks, e.g. <IMG> & </IMG> for the display of images, <P> & </P> for the insertion of a paragraph mark, and <CENTER> & </CENTER> to center the text .

For running Java applets on a Web page, you require the use of two special HTML tags: <APPLET> and <PARAM>. These tags are included on a Web page along with all other HTML code.

The applet tag is a surrounding tag. It may surround none or more parameter tags. It may also surround alternative text. Alternative text is text that appears between the <APPLET> and </APPLET> tags that is not included in a parameter tag. It is displayed by browsers that are not Java enabled as an alternative to the applet's display. The parameter tag is used to pass named parameters to a Java applet.

An applet uses the getParameter() method of the Applet class to retrieve the value of a parameter. The parameter tag may only appear between the <APPLET> and </APPLET> tags.

The <APPLET> and <PARAM> tags takes the following code:

```
<html>
<applet codebase=location of code code=filename.class
width=100 height=150 alt=alternate>
<param name="parameter" value="accepted value">
</applet>
</html>
```

Where CODE tag will take the name of executable java applet file, WIDTH will determine the width of the applet at runtime in the applet and HEIGHT will determine the height of the applet. CODEBASE will contain a reference to the directory or subdirectory where the applet and any related files will be found.

Consider the following HTML code.

```
<html>
<head> <title>My First Applet</title> </head>
```

```
<body>
<p>  Running First Applet <br></p>
<applet code="Newapp.class" width=400 height=200>
Sorry, Your browser doesn't support Java
</applet>
</body>
</html>
```

The above HTML code calls the applet class - Newapp.class in 400 by 200 rectangle. Consider the following code for Newapp.class

```
import java.awt.*;
public class Newapp extends java.applet.Applet{
Font f1=new Font("Arial",Font.BOLD,30);
public void paint(Graphics g){
    g.setFont(f1);
    g.setColor(Color.blue);
    g.drawString("SILICONMEDIA",10,40);
}
}
```

Note the usage of Font object - Font.BOLD. it has been declared as constant in the applet. Font class and blue is another constant declared in Color class. Two methods, setFont(Font) and setColor(Color) has been used to set the Font and Color of the Object respectively. The object is defined in the drawString() method. The output of the above HTML file is shown in Fig 6.2

Fig 6.2 Running Applet in Browser

ATTRIBUTES OF < APPLET> TAG

Align

You can use the ALIGN attribute to specify the alignment of applet's display region with respect to the rest of the line. This line may consist of text, images, or other HTML elements. This attribute can have values as TOP, TEXTTOP, BOTTOM, ABSBOTTOM, BASELINE, MIDDLE, ABSMIDDLE, LEFT, and RIGHT.

TOP	causes the top of an applet to be aligned with the top of the line being displayed by a browser.
TEXTTOP	causes the top of an applet to be aligned with the top of the text being displayed in the current line.
BASELINE	cause the bottom of the applet to be aligned with the baseline of the text in the line being displayed.
BOTTOM	same as BASELINE
ABSBOTTOM	causes the bottom of an applet to be aligned with the bottom of the current line being displayed.
MIDDLE	causes the middle of the applet to be aligned with the middle of the text displayed in the current line.
ABSMIDDLE	causes the middle of the applet to be aligned with the middle of the line being displayed.
LEFT and RIGHT	causes the applet to be aligned at the left and right margins of the browser window.

HSPACE and VSPACE

The HSPACE attribute specifies the number of pixels to be used as the left and right margins surrounding an applet. The VSPACE attribute specifies the number of pixels to be used as the top and bottom margins surrounding an applet.

PARAMETER TAG

The parameter tag is used to pass named parameters to a Java applet. It is a separating tag that has two attributes: NAME and VALUE. The NAME attribute identifies the name of a parameter and the VALUE attribute identifies its value.

Parameters are passed to the applet once it's loaded. All parameters are sent as strings whether or not they are encased in quotation marks.

The following is the example of the use of parameter tags:

<PARAM NAME="speed" VALUE="100">

The above example sends a parameter named speed with a value 100.

An applet can use the parameter only after retrieving the value of parameter. It uses getParameter() method of the Applet class to retrieve the value of a parameter. The parameter tag may only appear between the <APPLET> and </APPLET> tags.

Consider the following example, which displays the text at location specified using <PARAM> clause.

```
import java.applet.*;
import java.awt.*;
public class NewSample extends Applet {
String text = "No param parameter given";
int x = 0;
int y = 0;
public void init() {
    text = getParameter("text");
    try {
    x = Integer.parseInt(getParameter("x"));
    y = Integer.parseInt(getParameter("y"));
    }catch(NumberFormatException ex){
    }
    }
public void paint(Graphics g) {
    g.setFont(new
    Font("Arial",Font.BOLD+Font.ITALIC, 36));
    g.drawString(text,x,y);
    }
    }
```

The example is similar to the one discussed earlier but has been made more flexible, so that parameters can be defined using <PARAM> tag. The class uses the parseInt() method of the java.lang.Integer class to convert a String to an int as <PARAM> tag takes only the string values.(We will be discussing about such other method in the coming methods). The try and catch block is used to trap errors if the String cannot be converted to a number.

The HTML code can be written as follows:

```
<HTML>
<HEAD> <TITLE>My Second Applet</title> </HEAD>
<BODY>
<P>  Running Second Applet <BR></P>
<APPLET CODE="NewSample.class" HEIGHT=100 WIDTH=300>
<PARAM NAME="text" VALUE="Siliconmedia">
<PARAM NAME="x" VALUE="50">
```

```
<PARAM NAME="y" VALUE="50">
Sorry, Your browser doesn't support Java
</APPLET>
</HTML>
```

As you can see that the value of text, x and y has been defined using <PARAM> tag. The output of the abvoe HTML code is as shown below:

Fig 6.3 Running applet with variables

You may notice that we have not defined any stop() or destroy() methods. This is because we have no need to override either one of these method's activities. If you will move to a different Web page or otherwise dumps this applet page, the applet will stop and its resources will be recovered automatically.

EXERCISE

1. What type of argument is used with the paint() method?
   (a) A Graphics object
   (b) A Boolean variable
   (c) None
2. Which method is handled right before an applet finishes running?
   (a) decline()
   (b) destroy()
   (c) demise()
3. Why can't all variables needed in an applet be created inside the init() method?
   (a) The scope of the variables would be limited to the method only.
   (b) Federal legislation prohibits it.
   (c) They can be created there without any problems.
4. What's the name of the HTML tag that is used to send a parameter to a Java program?
   (a) <APPLET>
   (b) <VARIABLE>
   (c) <PARAM>
5. When the getParameter() method is used to load a parameter value into a program, what type of information is loaded?
   (a) a String variable
   (b) an array of characters
   (c) a different type depending on the parameter
6. If you try to load a parameter that is not included on the Web page that contains the applet, what will getParameter() do?
   (a) crash the program with an error
   (b) return the empty string null as a value
   (c) nothing
7. State True or False:
   1. When using applet viewer, the init() function of an applet is called only once when the applet is loaded.
   2. The start() method is executed everytime the control comes back to the browser window that displays the applet.
   3. The parameters passed to an applet can be of integer type.
   4. The applet execution doesn't depend on the browser setting.
   5. Applets cannot load programs from the machine where browser is displaying the applet.

6. Applets takes longer time to execute as compared to applications.

8. Can applets communicate with each other?

9. Can applets launch programs on the server?

10. Can applets launch programs on the client?

11. Is there a reason why the CODEBASE attribute should be used in an <APPLET> tag?

12. What happens if the height and width specified for an applet don't leave enough room for the information that is displayed in the paint() method?

THE STANDARD JAVA PACKAGES

THE LANGUAGE PACKAGE

THE OBJECT CLASS
THE CLASS CLASS
THE CLASSLOADER CLASS
WRAPPED CLASSES
THE MATH CLASS
THE STRING CLASS
THE STRINGBUFFER CLASS
THE SYSTEM CLASS
THE RUNTIME CLASS
THE CLOBNABLE CLASS
THE RUNNABLE CLASS

The Standard Java Packages

The standard Java Objects are the key to the application and applet programming. You turn this key and you have a wide variety of standard, reusable, inheritable classes, which save time and energy thus increasing the productivity. The Java standard packages contain groups of related classes. Along with classes, the standard Java packages also include interfaces, exception definitions, and error definitions. Java is composed of ten standard packages:

- Language package
- Utilities package
- I/O package
- Windowing package
- Text package
- Networking package
- Security package
- RMI package
- Reflection package
- SQL package

The following is the brief about these packages. Later in the chapter, we will take the first three packages in detail. The next couple of chapters will be covering Windows package.

The language package - java.lang, provides classes that make up the core of the Java language. The language package contains classes at the lowest level of the Java standard packages. Without this package, it is difficult to start the program. This package contains the core API classes of the JDK. It contains the following important classes :

- The Object class
- Data type classes (e.g Boolean, Character, String etc.)
- The Math class
- String classes
- System and Runtime classes
- Thread classes
- Class classes (Class and Classloader)
- Exception-handling classes
- The Process class

The utility package - java.util, provides various classes that perform different utility functions such as random number generation, dates, data structure class etc. The most important classes contained in the utilities package follow:

- The Date class
- Data structure classes
- The Random class
- The StringTokenizer class
- The Properties class
- Observer classes

The java.io - I/O package, provides classes that are required for reading and writing data to and from different input and output devices - including files. It includes package for input and output stream class hierarchy and stream filters to simplify I/O processing. It also has the package to perform random-access I/O and the StreamTokenizer class to construct input parses. The most important classes contained in the I/O package follow:

- Input stream classes
- Output stream classes
- File classes
- The StreamTokenizer class

The jawa.awt - abstract windows toolkit, or windows packaging is the base for for Java window programming. This package includes classes representing graphical interface elements such as windows, dialog boxes, menus, buttons, checkboxes, scrollbars, and text fields, as well as general graphics elements such as fonts. The most important classes included in the windowing package are as follow:

- Graphical classes
- Layout manager classes
- Font classes
- Dimension classes
- The MediaTracker class

The Java networking package, also known as java.net, contains classes that allow you to perform a wide range of network communications. It includes classes which improves the productivity while developing client/server applications. The java.net package also provides a set of classes that support network programming using the communication protocols employed by the Internet. These protocols are known as the Internet protocol suite and include the Internet Protocol (IP), the Transport Control Protocol (TCP), and the User Datagram Protocol (UDP) as well as other, less-prominent supporting protocols. The classes contained in the Networking package are as follow:

- The InetAddress Class
- The Socket Class
- The ServerSocket Class
- The DatagramSocket Class
- The DatagramPacket Class
- The SocketImpl Class
- The SocketImplFactory Interface
- Web-Related Classes
- The ContentHandler and ContentHandlerFactory Classes
- The URLStreamHandler Class
- The URLStreamHandlerFactory Interface

The InetAddress class encapsulates Internet IP addresses and supports conversion between dotted decimal addresses and hostnames.

The Socket, ServerSocket, and DatagramSocket classes implement client and server sockets for connection-oriented and connectionless communication. The SocketImpl class and the SocketImplFactory interface provide hooks for implementing custom sockets.

The URL, URLConnection, and URLEncoder classes implement high-level browser-server Web connections. The ContentHandler and URLStreamHandler classes are abstract classes that provide the basis for the implementation of Web content and stream handlers. They are supported by the ContentHandlerFactory and URLStreamHandlerFactory interfaces.

java.text, the text package, contains the classes that are used for internationalization. The text package contains classes and interfaces for handling text specific to a particular locale. The text package provides the classes that enables text to be mapped to a specific language and region. The classes and interfaces defined in the text package rely on Unicode 2.0 character encoding and can be used to adapt text, numbers, dates, currency, and user-defined objects to the conventions of any country. Some of the more important classes included in the text package follow:

- Formatting classes
- The Collator class
- The TextBoundary class

The Java security package, also known as java.security, includes classes for incorporating cryptographic security into Java-based applications. It includes support for DSA cryptography)built-in digital signature) and is designed so that new algorithms can be

added later without difficulty. The few important classes included in the package are :

- Digital signature classes
- The MessageDigest class
- Key management classes

The Java RMI (Remote Method Invocation) package - java.rmi, lets you create distributed Java-to-Java applications that rely on remote method invocation. Using RMI, you can invoke methods of remote Java objects from other Java virtual machines, including different hosts.

The Java reflection package - java.lang.reflect, is an extension of language package, helps to find detailed information regarding the structure of classes at runtime. Also this package can also be used to discover information about the fields, methods, and constructors of classes. This package also includes applications that require access to either the public members of a target object or the members declared by a given class.

The Java SQL package, - java.sql (also known as JDBC package) , lets you develop database applications capable of performing SQL queries. Using SQL package, you can make a Java application to interact with virtually any relational database using SQL. The few important classes and interfaces included in the SQL package follow:

- The DriverManager class
- The Connection interface
- The Statement and ResultSet interfaces

The Language Package

The Java language package is the heart of Java API. It provides the core classes that make up the Java programming environment. The language package includes classes representing numbers, strings, and objects, as well as classes for handling compilation, the runtime environment, security, and threaded programming. Unlike other packages, you don't have to import the java.lang package, it is imported automatically into every Java program.

We have already listed the various important classes of java.lang package. Here, you will find the detailed description of these classes.

THE OBJECT CLASS

Object class is the most important of all classes as it is the Superclass of all Java classes. The Object class does not have any variables and has only one constructor. However, it provides 11 methods that are inherited by all Java classes and that support general operations that are used with all objects.

The definition for the Object class follows:

```
public class java.lang.Object {
// Constructors
public Object();
// Methods
protected Object clone();
public boolean equals(Object obj);
protected void finalize();
public final Class getClass();
public int hashCode();
public final void notify();
public final void notifyAll();
public String toString();
public final void wait();
public final void wait(long timeout);
public final void wait(long timeout, int nanos);
}
```

This method creates a clone of this object by creating a new instance of the class and copying each of the member variables of this object to the new object. The object must implement the Cloneable interface. This interface is defined within the java.lang package. It contains no methods and is used only to differentiate cloneable from noncloneable classes. The definition for the **loneable** interface follows:

```
public interface java.lang.Cloneable {
}
```

It could throws the following two exceptions:

OutOfMemoryError	if there is not enough memory.
CloneNotSupportedException	if the object doesn't support the Cloneable interface or if it explicitly doesn't want to be cloned.

The equals() and hashCode() methods are used to construct hash tables of Java objects. Hash tables are like arrays, but they are indexed by key values and dynamically grow in size. They make use of hash functions to quickly access the data that they contain. The hashCode() method creates a hash code for an object. Hash codes are used to quickly determine whether two objects are different. It will be discussed in more details in the next section - java.util package.

The getClass() method identifies the class of an object by returning an object of Class. The Class object keeps up with runtime class information such as the name of a class and the parent superclass.

This method determines a string representation of this object. It is recommended that all derived classes override toString as the value of an object varies depending on the class type. The information returned by toString() can be very valuable for determining the internal state of an object when you are debugging your code.

As already discussed in the previous chapters, the finalize() method of an object is executed when an object is garbage-collected. The method performs no action, by default, and needs to be overridden by any class that requires specialized finalization processing.

This method wakes up a single thread that is waiting on this object's monitor. A thread is set to wait on an object's monitor when the wait method is called. The notify method should be called only by a thread that is the owner of this object's monitor. The notifyall() method wakes up all threads that are waiting on this object's monitor.

These methods could throws the following exceptions :

IllegalMonitorStateException	if the current thread is not the owner of this object's monitor.

This method causes the current thread to wait. wait() for forever and wait(long timeout) or wait(long timeout, int nanos)until it is notified via a call to the notify or notifyAll method, or until the specified timeout period has elapsed. The wait(long timeout, int nanos) lets you define the timeout period in nanos for a finer control.

All these methods could throw following two exceptions:

IllegalMonitorStateException	if the current thread is not the owner of this object's monitor.
InterruptedException	if another thread has interrupted this thread.

THE CLASS CLASS

This class implements a runtime descriptor for classes and interfaces in a running Java program. It provides eight methods that support the runtime processing of an object's class and interface information. This class does not have a constructor. Instances of Class, referred to as class descriptors, are constructed automatically by the Java virtual machine when classes are loaded, which explains why there are no public constructors for the class. The definition for the Class class follows:

```
public final class java.lang.Class extends java.lang.Object
{
// Methods
public static Class forName(String className);
static Class forName(String name, boolean initialize,
                     ClassLoader loader);
public Class[] getClasses();
public ClassLoader getClassLoader();
public Class getComponentType();
public Constructor getConstructor(Class[]
                     parameterTypes);
public Constructor[] getConstructors();
public Class[] getDeclaredClasses();
public Constructor getDeclaredConstructor(Class[]
                     parameterTypes);
public Constructor[] getDeclaredConstructors();
public Field getDeclaredField(String name) Field[]
                     getDeclaredFields();
public Method getDeclaredMethod(String name, Class[]
                     parameterTypes);
public Method[] getDeclaredMethods();
public Class getDeclaringClass();
public Field getField(String name);
```

```
public Field[] getFields();
public Class[] getInterfaces();
public Class[] getInterfaces();
public Method getMethod(String name, Class[]
                        parameterTypes);
public Method[] getMethods();
public int getModifiers();
public String getName();
public Package getPackage();
public ProtectionDomain getProtectionDomain();
public URL getResource(String name);
public InputStream getResourceAsStream(String name)
                                  Object[] getSigners();
public Class getSuperclass();
public boolean isArray();
public boolean isAssignableFrom(Class cls);
public boolean isInstance(Object obj);
public boolean isInterface();
public boolean isPrimitive();
public Object newInstance();
public String toString();
}
```

Given below is the description for various important methods:

This method determines the runtime class descriptor for the class with the specified name. E.g. forName() method can be used to get information about the Float class:

```
Class info = Class.forName("java.lang.Float");
```

It can throw the following exception :

ClassNotFoundException if the class could not be found.

Returns an array containing Class objects representing all the public classes and interfaces that are members of the class represented by this Class object.

Returns the class loader for the class.

Returns an array containing Constructor objects reflecting all the public constructors of the class represented by this Class object.

Returns a Field object that reflects the specified public member field of the class or interface represented by this Class object.

This method determines the interfaces implemented by the class or interface represented by this object.

Returns the Java language modifiers for this class or interface, encoded in an integer.

This method determines the fully qualified name of the class or interface represented by this object.

e.g. String sm = info.getName()

Gets the package for this class.

Gets the signers of this class.

This method determines the superclass of the class represented by this object.

Determines if this Class object represents an array class.

Determines if the class or interface represented by this Class object is either the same as, or is a superclass or superinterface of, the class or interface represented by the specified Class parameter.

Determines if the specified Object is assignment-compatible with the object represented by this Class.

This method determines whether the class represented by this object is actually an interface.

Determines if the specified Class object represents a primitive type.

This method creates a new default instance of the class represented by this object. It can be used in place of a class's constructor, although it is generally safer and clearer to use a constructor rather than newInstance().

It could throws the following two exceptions:

InstantiationException	if you try to instantiate an abstract class or an interface, or if the instantiation fails for some other reason.
IllegalAccessException	if the class is not accessible.

This method determines the name of the class or interface represented by this object, with the string "class" or the string "interface" prepended appropriately. The toString() method differs from toString() method of Object class, in that it prepends the string class or interface, depending on whether the class descriptor is a class or an interface.

This method determines the class loader for this object.

THE CLASSLOADER CLASS

By default, the runtime system loads classes from files in the directory defined in the CLASSPATH environment variable. Classes that are loaded from outside CLASSPATH require a class loader to convert the class byte stream into a class descriptor. ClassLoader is an abstract class that is used to define class loaders.

This is a platform-dependent process and doesn't involve ClassLoader objects. The ClassLoader class comes into play when you want to define other techniques of loading classes, such as across a network connection. Following are the important methods of classloader class:

```
protected final Class defineClass(byte data[], int off, int
len);
protected final Class findSystemClass(String name);
protected abstract Class loadClass(String name, boolean
resolve);
protected final void resolveClass(Class c);
}
```

This constructor creates a default class loader. If a security manager is present, it is checked to see whether the current thread has

permission to create the class loader. If not, a SecurityException is thrown.

SecurityException	if the current thread doesn't have permission to create the class loader.

This method converts an array of bytes into an instance of class Class by reading len bytes from the array b beginning off bytes into the array. Here, b is the byte array containing the class data, off is the starting offset into the array for the data and len is the length in bytes of the class data.

It could throw the following exception:

ClassFormatError	if the class data does not define a valid class.

This method loads the class with the specified name, resolving it if the resolve parameter is set to TRUE. This method must be implemented in all derived class loaders, because it is defined as abstract. It can also throw the following exception:

ClassNotFoundException	if the class is not found.

This method resolves the specified class so that instances of it can be created or so that its methods can be called.

This method finds the system class with the specified name, loading it if necessary. A system class is a class loaded from the local file system with no class loader in a platform-specific manner. It could throw the following exceptions:

ClassNotFoundException	if the class is not found.
NoClassDefFoundError	if a definition for the class is not found.

WRAPPED CLASSES

Object type wrappers are useful because many Java classes operate on objects rather than primitive data types. Furthermore, by creating object versions of the simple data types, it is possible to add useful member functions for each data type.

The Boolean class is a wrapper for the boolean primitive type. It provides support constants and methods for working with Boolean values. The important methods for the Boolean class are as follows:

The constant **oolean** object can represent the primitive Boolean value - TRUE or FALSE. It has two constructors and six methods. The Boolean(boolean value) constructor creates a Boolean wrapper object representing the specified primitive Boolean value whereas Boolean(String s) constructor creates a Boolean wrapper object representing the specified string "TRUE" or "FALSE". The methods are defined below.

```
public boolean booleanValue();
public boolean equals(Object obj);
public static boolean getBoolean(String name);
public int hashCode();
public String toString();
public static Boolean valueOf(String s);
```

booleanValue()	determines the primitive Boolean value represented by this object.
equals()	compares the Boolean value of the specified object to the Boolean value of this object.
getBoolean()	determines the Boolean value of the system property with the specified name.
HashCode()	calculates a hash code for this object.
toString()	determines a string representation of the primitive Boolean value for this object.
valueOf()	creates a new Boolean wrapper object based on the Boolean value represented by the specified string.

This class implements an object type wrapper for character values. It provides several methods that support case, type, and class testing, and conversion. The definition for the Character class follows:

```
public final class java.lang.Character extends
java.lang.Object {
// Member Constants
public final static int MAX_RADIX;
public final static char MAX_VALUE;
public final static int MIN_RADIX;
public final static char MIN_VALUE;
// Constructors
public Character(char value);
// Methods
public char charValue();
public int compareTo(Character anotherCharacter)
public static int digit(char ch, int radix);
public boolean equals(Object obj);
public static char forDigit(int digit, int radix);
public static int getNumericValue(char ch)
```

```
public static int getType(char ch)
public int hashCode();
public static boolean isDefined(char ch);
public static boolean isDigit(char ch);
public static boolean isIdentifierIgnorable(char ch)
public static boolean isISOControl(char ch)
public static boolean isJavaIdentifierPart(char ch)
public static boolean isJavaIdentifierStart(char ch)
public static boolean isLetter(char ch);
public static boolean isLetterOrDigit(char ch);
public static boolean isLowerCase(char ch);
public static boolean isTitleCase(char ch);
public static boolean isUnicodeIdentifierPart(char ch)
public isUnicodeIdentifierStart(char ch)
public static boolean isUpperCase(char ch);
public static char toLowerCase(char ch);
public static boolean isWhitespace(char ch)
public String toString();
public static char toTitleCase(char ch);
public static char toUpperCase(char ch);
}
```

Following the description of important methods:

charValue() determines the primitive character value represented by this object.

compareTo(Character anotherCharacter)
Compares two Characters numerically.

digit() determines the numeric value of the specified character digit using the specified radix. (MAX_RADIX = 36, MIN_RADIX = 2).

equals() compares the character value of the specified object to the character value of this object.

forDigit() determines the character value of the specified numeric digit using the specified radix.

getNumericValue(char ch)
Returns the Unicode numeric value of the character as a nonnegative integer.

getType(char ch) Returns a value indicating a character category.

hashCode() calculates a hash code for this object.

isDefined() determines whether the specified character has a defined Unicode meaning.

isDigit() determines whether the specified character is a numeric digit.

isLetter() determines whether the specified character is a letter.

isLetterOrDigit() determines whether the specified character is a letter or digit.

isLowerCase() determines whether the specified character is a lowercase character.

isTitleCase() determines whether the specified character is a titlecase character.

isUpperCase() determines whether the specified character is an uppercase character.

isWhitespace(char ch)

Determines if the specified character is white space according to Java.

toLowerCase() converts the specified character to a lowercase character.

toString() determines a string representation of the primitive character value for this object.

toTitleCase() converts the specified character to a titlecase character.

toUpperCase() converts the specified character to an uppercase character.

This class is an abstract class that provides the basic functionality required of a numeric object. It that is subclassed by Integer, Long, Float, and Double. All specific numeric objects are derived from Number. The definition for the Number class follows:

```
public abstract class java.lang.Number extends
java.lang.Object {
// Methods
public abstract double doubleValue();
public abstract float floatValue();
public abstract int intValue();
public abstract long longValue();
}
```

doubleValue() determines the primitive double value represented by this object.

floatValue() determines the primitive float value represented by this object.

intValue() determines the primitive integer value represented by this object.

longValue() determines the primitive long value represented by this object.

The Byte class wraps the fundamental byte type and provides a variety of methods for working with byte numbers. Some of the more important methods implemented by Byte follow:

```
static int parseByte(String s, int radix);
static int parseByte(String s);
long shortValue();
long longValue();
long intValue();
float floatValue();
double doubleValue();
```

parseByte() parse strings for a byte value and return the value as a byte.

int/short/long/float/doubleValue() return the value of a byte converted to the appropriate type.

This class implements an object type wrapper for integer values. Object type wrappers are useful because many Java classes operate on objects rather than primitive data types. In addition, the Integer class provides support constants and methods for working with integer values. The methods implemented by Integer follow:

```
static int parseInt(String s, int radix);
static int parseInt(String s);
short shortValue();
long longValue();
float floatValue();
double doubleValue();
byte byteValue();
static Integer getInteger(String name);
static Integer getInteger(String name, int val);
static Integer getInteger(String name, Integer val);
```

parseInt() parse strings for an integer value and return the value as an int.

getInteger() return an integer property value specified by the String property name parameter name.

The Long class is similar to the Integer class except that it wraps the fundamental type long. The Long class implements methods similar to those of the Integer class, with the exception that they act on long-type numbers rather than on int-type numbers.

This class implements an object type wrapper for float values. Object type wrappers are useful because many Java classes operate on objects rather than primitive data types. In addition, the Float class provides support constants and methods for working with float values. The methods implemented by the Float class follow:

```
boolean isNaN();
static boolean isNaN(float v);
boolean isInfinite();
static boolean isInfinite(float v);
short shortValue();
long longValue();
int intValue();
double doubleValue();
byte byteValue();
static int floatToIntBits(float value);
static float intBitsToFloat(int bits);
```

isNaN()	returns whether or not the Float value is the special not-a-number (NaN) value.
isInfinite()	returns whether or not the Float value is infinite, which is represented by the special NEGATIVE_INFINITY and POSITIVE_INFINITY final static member constants.
floatToIntBits()	determines the IEEE 754 floating-point single precision representation of the specified float value.
IntBitsToFloat()	determines the float representation of the specified IEEE 754 floating-point single precision value.

```
import java.lang.System;
import java.lang.Boolean;
import java.lang.Character;
import java.lang.Integer;
import java.lang.Long;
import java.lang.Float;
import java.lang.Double;
public class wrapp{
    public static void main(String args[]){
    Boolean b1 = new Boolean("TRUE");
    Boolean b2 = new Boolean("FALSE");
    System.out.println(b1.toString()+" or"+b2.toString());
    for(int j=0;j<16;++j)
            System.out.print(Character.forDigit(j,16));
            System.out.println();
            Integer i = new Integer(Integer.parseInt("ef",16));
```

```
                Long l = new Long(Long.parseLong("abcd",16));
                long m=l.longValue()*i.longValue();
                System.out.println(Long.toString(m,8));
                System.out.println(Float.MIN_VALUE);
                System.out.println(Double.MAX_VALUE);
        }
    }
```

The output of the above program is

```
TRUE or FALSE
0123456789abcdef
50062143
1.4E-45
1.7976931348623157E308
```

THE MATH CLASS

The Math class contains many invaluable mathematical functions along with a few useful constants. This class implements a library of common math functions, including methods for performing basic numerical operations such as elementary exponential, logarithm, square root, and trigonometric functions. Additionally, the Math class is declared as final, i.e. you can't derive from it. The definition for the Math class is as follows:

```
public final class java.lang.Math extends java.lang.Object
{
// Member Constants
public final static double E;
public final static double PI;
// Methods
public static double abs(double a);
public static float abs(float a);
public static int abs(int a);
public static long abs(long a);
public static double acos(double a);
public static double asin(double a);
public static double atan(double a);
public static double atan2(double a, double b);
public static double ceil(double a);
public static double cos(double a);
public static double exp(double a);
public static double floor(double a);
public static double IEEEremainder(double f1, double f2);
public static double log(double a);
public static double max(double a, double b);
public static float max(float a, float b);
```

```
public static int max(int a, int b);
public static long max(long a, long b);
public static double min(double a, double b);
public static float min(float a, float b);
public static int min(int a, int b);
public static long min(long a, long b);
public static double pow(double a, double b);
public static double random();
public static double rint(double a);
public static long round(double a);
public static int round(float a);
public static double sin(double a);
public static double sqrt(double a);
public static double tan(double a);
public static double toDegrees(double angrad);
public static double toRadians(double angdeg);
}
```

abs() calculates the absolute value of the specified double value.

acos() calculates the arccosine of the specified double value.

asin() calculates the arcsine of the specified double value.

atan() calculates the arctangent of the specified double value.

atan2() calculates the theta component of the polar coordinate (r, theta) corresponding to the rectangular coordinate (x, y) specified by the double values.

ceil() determines the smallest double whole number that is greater than or equal to the specified double value.

Cos() calculates the cosine of the specified double value, which is specified in radians.

Exp() calculates the exponential value of the specified double value, which is E raised to the power of a.

floor() determines the largest double whole number that is less than or equal to the specified double value.

IEEEremainder() calculates the remainder of f1 divided by f2, as defined by the IEEE 754 standard.

log() calculates the natural logarithm (base E) of the specified double value.

max() determines the larger of the two specified double values.

min() determines the smaller of the two specified double values.

pow()	calculates the double value a raised to the power of b.
random()	generates a pseudo-random double between 0.0 and 1.0.
rint()	determines the closest whole number to the specified double value.
round()	rounds off the specified double value by determining the closest long value.
sin()	calculates the sine of the specified double value, which is specified in radians.
sqrt()	calculates the square root of the specified double value.
tan()	calculates the tangent of the specified double value, which is specified in radians.
toDegrees()	Converts an angle measured in radians to the equivalent angle measured in degrees.
toRadians()	Converts an angle measured in degrees to the equivalent angle measured in radians.

```
import java.lang.System;
import java.lang.Math;
public class MathApp {
public static void main(String args[]) {
System.out.println(Math.E);
System.out.println(Math.PI);
System.out.println(Math.abs(-1234));
System.out.println(Math.cos(Math.PI/4));
System.out.println(Math.sin(Math.PI/2));
System.out.println(Math.tan(Math.PI/4));
System.out.println(Math.log(1));
System.out.println(Math.exp(Math.PI));
System.out.println(Math.sqrt(49.00));
System.out.println(Math.pow(2,4));
for(int i=0;i<5;++i){
    System.out.print(Math.random());
    System.out.println(\n);
    }
System.out.println();
}
}
```

The output of the above program is

```
2.718281828459045
3.141592653589793
1234
6.123031769111886E-17
```

```
1.0
0.9999999999999999
0.0
23.140692632779267
7.0
16.0
0.7474941425264944
0.3570842695525769
0.81586852214149
0.5440026029429945
0.4635666300132165
```

THE STRING CLASS

This class implements a constant string of characters. The String class provides a wide range of support for working with strings of characters. The String class supports constant (unchanging) strings. Note that literal string constants are converted automatically to String objects by the Java compiler. The definition for the String class follows:

```
public final class java.lang.String extends java.lang.Object
{
// Constructors
public String();
public String(byte ascii[], int hibyte);
public String(byte ascii[], int hibyte, int off, int count);
public String(char value[]);
public String(char value[], int off, int count);
public String(String value);
public String(StringBuffer buffer);
// Methods
public char charAt(int index);
public int compareTo(Object o)
public int compareTo(String anotherString);
public int compareToIgnoreCase(String str)
public String concat(String str);
public static String copyValueOf(char data[]);
public static String copyValueOf(char data[], int off, int
count);
public boolean endsWith(String suffix);
public boolean equals(Object anObject);
public boolean equalsIgnoreCase(String anotherString);
public void getBytes();
public void getChars(int srcBegin, int srcEnd, char dst[],
int stBegin);
```

```
public int hashCode();
public int indexOf(int ch);
public int indexOf(int ch, int fromIndex);
public int indexOf(String str);
public int indexOf(String str, int fromIndex);
public String intern();
public int lastIndexOf(int ch);
public int lastIndexOf(int ch, int fromIndex);
public int lastIndexOf(String str);
public int lastIndexOf(String str, int fromIndex);
public int length();
public boolean regionMatches(boolean ignoreCase, int
toffset, String other, int ooffset, int len);
public boolean regionMatches(int toffset, String other,
int ooffset, int len);
public String replace(char oldChar, char newChar);
public boolean startsWith(String prefix);
public boolean startsWith(String prefix, int toffset);
public String substring(int beginIndex);
public String substring(int beginIndex, int endIndex);
public char[] toCharArray();
public String toLowerCase();
public String toLowerCase(Locale locale);
public String toString();
public String toUpperCase();
public String toUpperCase(Locale locale);
public String trim();
public static String valueOf(boolean b);
public static String valueOf(char c);
public static String valueOf(char data[]);
public static String valueOf(char data[], int off, int count);
public static String valueOf(double d);
public static String valueOf(float f);
public static String valueOf(int i);
public static String valueOf(long l);
public static String valueOf(Object obj);
}
```

charAt() determines the character at the specified index. Returns the following exception:

StringIndexOutOfBoundsException if the index is out of range.

compareTo() compares this string with the specified string lexicographically.

concat()	concatenates the specified string onto the end of this string.
copyValueOf()	converts a character array to an equivalent string by creating a new string and copying the characters into it.
endsWith()	determines whether this string ends with the specified suffix.
equals()	compares the specified object to this string.
equalsIgnoreCase()	compares the specified string to this string, ignoring case.
getBytes()	Convert this String into bytes according to the platform's default character encoding, storing the result into a new byte array.
getChars()	copies each character in this string, beginning at srcBegin and ending at srcEnd, into the character array dst beginning at dstBegin. Throws **StringIndexOutOfBoundsException** exception.
hashCode()	calculates a hash code for this object.
indexOf()	determines the index of the first occurrence of the specified character in this string.
intern()	determines a string that is equal to this string but is guaranteed to be from a pool of unique strings.
lastIndexOf()	determines the index of the last occurrence of the specified character in this string.
length()	determines the length of this string, which is the number of Unicode characters in the string.
regionMatches()	determines whether a substring of this string matches a substring of the specified string, with an option for ignoring case.
replace()	replaces all occurrences of oldChar in this string with newChar.
startsWith()	determines whether this string starts with the specified prefix.
substring()	determines the substring of this string, beginning at beginIndex.
toCharArray()	converts this string to a character array by creating a new array and copying each character of the string to it.
toLowerCase()	converts all the characters in this string to lowercase.
toString()	returns this string.
toUpperCase()	converts all the characters in this string to uppercase.
trim()	trims leading and trailing whitespace from this string.

valueOf() creates a string representation of the specified Boolean value.

```
import java.lang.System;
import java.lang.String;
public class strapp {
public static void main(String args[]) {
String s = "SILICONMEDIA.org";
System.out.println(s);
System.out.println(s.toUpperCase());
System.out.println(s.toLowerCase());
System.out.println("["+s+"]");
s=s.trim();
System.out.println("["+s+"]");
s=s.replace( S , P );
s=s.replace( M , Q );
s=s.replace( A , R );
System.out.println(s);
int i1 = s.indexOf( P );
int i2 = s.indexOf( Q );
int i3 = s.indexOf( R );
char ch[] = s.toCharArray();
ch[i1]= S ;
ch[i2]= M ;
ch[i3]= A ;
s = new String(ch);
System.out.println(s);
}
}
```

The output of the following program would be:

```
SILICONMEDIA.org
SILICONMEDIA.ORG
siliconmedia.org
[SILICONMEDIA.org]
[SILICONMEDIA.org]
PILICONQEDIR.org
SILICONMEDIA.org
```

THE STRINGBUFFER CLASS

This class implements a variable string of characters. The StringBuffer class provides a wide range of append and insert methods, along with some other support methods for getting information about the string

buffer. Note that the StringBuffer class is synchronized appropriately so that it can be used by multiple threads. The basic difference between String and StringBuffer class is that the String class supports constant (unchanging) strings, whereas the StringBuffer class supports growable, modifiable strings. Also, String objects are more compact than StringBuffer objects, but StringBuffer objects are more flexible. Following are the various important methods:

```
public StringBuffer append(boolean b);
public StringBuffer append(char c);
public StringBuffer append(char str[]);
public StringBuffer append(char str[], int off, int len);
public StringBuffer append(double d);
public StringBuffer append(float f);
public StringBuffer append(int i);
public StringBuffer append(long l);
public StringBuffer append(Object obj);
public StringBuffer append(String str);
public int capacity();
public char charAt(int index);
public void ensureCapacity(int minimumCapacity);
public void getChars(int srcBegin, int srcEnd, char dst[],
int dstBegin);
public StringBuffer insert(int off, boolean b);
public StringBuffer insert(int off, char c);
public StringBuffer insert(int off, char str[]);
public StringBuffer insert(int off, double d);
public StringBuffer insert(int off, float f);
public StringBuffer insert(int off, int i);
public StringBuffer insert(int off, long l);
public StringBuffer insert(int off, Object obj);
public StringBuffer insert(int off, String str);
public int length();
public StringBuffer reverse();
public void setCharAt(int index, char ch);
public void setLength(int newLength);
public String toString();
}
```

StringBuffer class provides following methods, different from the String Class.

append()	appends the string representation of the specified value/object to the end of this string buffer.
capacity()	determines the capacity of this string buffer.

ensureCapacity() ensures that the capacity of this string buffer is at least equal to the specified minimum.

insert() method inserts the string representation of the specified value at the specified offset of this string buffer.

setCharAt() changes the character at the specified index in this string to the specified character.

setLength() explicitly sets the length of this string buffer.

```
import java.lang.System;
import java.lang.String;
import java.lang.StringBuffer;
public class strbpp {
public static void main(String args[]) {
StringBuffer s1 = new StringBuffer(" Book s are ");
s1.append("Cool");
s1.append( ! );
s1.insert(0,"Siliconmedia");
s1.append( \n );
s1.append("This is ");
s1.append(true);
s1.setCharAt(38, T );
s1.append( \n );
s1.append("Aiming at #");
s1.append(1);
String s = s1.toString();
System.out.println(s);
}
}
```

The output of the above code is :

```
Siliconmedia Book s are Cool!
This is True
Aiming at #1
```

THE SYSTEM CLASS

This class provides a platform-independent means of interacting with the Java runtime system. The System class provides support for standard input, standard output, and standard error streams, along with providing access to system properties, among other things. Note that the System class cannot be instantiated or subclassed because all its methods and variables are static. Following are the various important methods of StringBuffer class:

```
public static void arraycopy(Object src, int
src_position, Object dst,int dst_position, int length);
public static long currentTimeMillis();
public static void exit(int status);
public static void gc();
public static Properties getProperties();
public static String getProperty(String key);
public static String getProperty(String key, String def);
public static SecurityManager getSecurityManager();
public static void load(String pathname);
public static void loadLibrary(String libname);
public static void runFinalization();
public static void setProperties(Properties props);
public static void setSecurityManager(SecurityManager
s);
```

Given below is the definition of various methods:

arraycopy()	copies len array elements from the src array, beginning at src_position, to the dst array, beginning at dst_position. Throws **ArrayStoreException** if an element in the source array could not be stored in the destination array due to a type mismatch.
currentTimeMillis()	determines the current UTC time relative to midnight, January 1, 1970 UTC, in milliseconds.
exit()	exits the Java runtime system (virtual machine) with the specified integer exit status. Throws **SecurityException** if the current thread cannot exit with the specified exit status.
gc()	invokes the Java garbage collector to clean up any objects that no longer are needed.
getProperties()	determines the current system properties. Throws **SecurityException** if the current thread cannot access the system properties.
loadLibrary()	loads the dynamic library with the specified library name. Throws **UnsatisfiedLinkError** if the library doesn't exist and **SecurityException** if the current thread cannot access the system properties.
runFinalization()	explicitly causes the finalize methods of any discarded objects to be called.
setProperties()	sets the system properties to the specified properties.

THE RUNTIME CLASS

This class provides a mechanism for interacting with the Java runtime environment. Each running Java application has access to a single instance of the Runtime class, which it can use to query and

modify the runtime environment. Note that Runtime objects cannot be created directly by a Java program. The definition for the Runtime class follows:

```
public class java.lang.Runtime extends java.lang.Object
{
// Methods
public void addShutdownHook(Thread hook)
public Process exec(String command);
public Process exec(String command, String envp[]);
public Process exec(String cmdarray[]);
public Process exec(String cmdarray[], String envp[]);
public Process exec(String[] cmdarray, String[] envp,
File dir)
public Process exec(String cmd, String[] envp)
public Process exec(String command, String[] envp, File
dir)
public void exit(int status);
public long freeMemory();
public void gc();
public static Runtime getRuntime();
public void halt(int status);
public void load(String filename);
public void loadLibrary(String libname);
public boolean removeShutdownHook(Thread hook);
public void runFinalization();
public long totalMemory();
public void traceInstructions(boolean on);
public void traceMethodCalls(boolean on);
}
```

Following is the definition for various important methods:

exec()	executes the system command represented by the specified string in a separate subprocess.
freeMemory()	determines the approximate amount of free memory available in the runtime system, in bytes.
getRuntime()	gets the runtime environment object associated with the current Java program.
load()	loads the dynamic library with the specified complete path name. Throws **UnsatisfiedLinkError** if the library doesn't exist. **SecurityException** if the current thread can't load the library.
totalMemory()	determines the total amount of memory in the runtime system, in bytes.

Write a program to find out the total memory and the free memory in your computer. Extend the program to execute the to applications - Windows Browser and Notepad.

Solution:

```
import java.lang.System;
import java.lang.Runtime;
import java.io.IOException;
public class RuntimeMemApp {
public static void main(String args[]) throws
IOException {
    Runtime rt = Runtime.getRuntime();
    System.out.println(rt.totalMemory());
    System.out.println(rt.freeMemory());
    rt.exec("C:\\Windows\\Explorer.exe");
    rt.exec("C:\\Windows\\Notepad.exe");
}
}
```

THE CLOBNABLE CLASS

This interface indicates that an object can be cloned using the clone method defined in Object. The clone method clones an object by copying each of its member variables. It throws **CloneNotSupportedException** if the object doesn't support the Cloneable interface or if it explicitly doesn't want to be cloned. The definition for the Cloneable interface follows:

```
public interface java.lang.Cloneable {
    }
```

THE RUNNABLE CLASS

This interface provides a means for an object to be executed within a thread without having to be derived from the Thread class. We have already discussed this class in the Chapter "Multithreading". The definition for the Runnable interface follows:

```
public interface java.lang.Runnable {
    public abstract void run();
    }
```

EXERCISE

1. What are the various important packages, also mention the various classes of the important packages?
2. What is the significance of toString() method in Object Class?
3. Differentiate between String and StringBuffer Class.
4. Fill in the Blanks:
   1. _____ method determines the current system properties whereas _____ method sets system properties to the specified properties.
   2. _____ method creates clone of this object by creating a new instance of the class.
   3. _____ method returns the loader of the class.
   4. _____ method determines if this Class object represents an array class.
   5. IllegalAccessException is thrown by _____ method in java.lang.
   6. _____ method determines the approximate amount of free memory available in the runtime system.
   7. _____ method determines the length of this string.
   8. _____ method determines whether the specified character is a numeric digit.

THE UTILITY PACKAGE

INTERFACES
THE DATE CLASS
THE BITSET CLASS
THE CALENDER CLASS
THE DICTIONARY CLASS
THE HASHTABLE CLASS
THE PROPERTIES CLASS
THE VECTOR CLASS
THE OBSERVABLE CLASS
THE STACK CLASS
THE STRINGTOKENIZER CLASS
THE RANDOM CLASS

The Utility Package

The Utility package - java.util, contains the classes that implement many of those features or functions usually left for the programmer or someone else to implement. Here, you'll learn to use the Date class to manipulate Date objects, to generate random numbers using the Random class, and to work with data structures such as dictionaries, stacks, hash tables, vectors, and bit sets. These classes are also useful in a variety of other ways and are the fundamental building blocks of the more complicated data structures used in other Java packages and in your own applications.

The classes contained in the utilities package are given below:

- The Date class
- Data structure classes
- The Random class
- The StringTokenizer class
- The Properties class
- Observer classes
- The Enumeration classes
- Hashtable classes

INTERFACES

The interface java.util.Enumeration defines methods that can be used to iterate through a set of objects. The methods hasMoreElements and nextElement are typically used in a loop that visits each item in the set.

```
public interface Enumeration {
    // public instance methods
    public abstract boolean hasMoreElements();
    public abstract Object nextElement();
    }
```

hasMoreElements() used to determine if the enumeration has more elements.

nextElement() returns the next element in the enumeration. Calling it repeatedly will move through the enumeration. Throws **NoSuchElementException** is thrown if there are no more elements in the enumeration.

For example, You can use an Enumeration object to print all the elements of a Vector object, v, as follows:

```
for (Enumeration e=v.elements();e.hasMoreElements();)
   System.out.print(e.nextElement()+" ");
```

The interface java.util.Observer defines an update method that is invoked by an Observable object whenever the Observable object has changed and wants to notify its Observers.

```
public interface Observer {
// public instance methods
public abstract void update(Observable o, Object arg);
}
```

update() This method is called whenever an Observable instance that is being observed invokes either of its **notifyObservers** methods.

THE DATE CLASS

The class java.util.Date extends Object. The Date class stores a representation of a date and time and provides methods for manipulating the date and time components. A new Date instance may be constructed using any of the following:

- The current date and time as expressed in the UNIX-standard milliseconds since midnight January 1, 1970;
- A String;
- Integers representing the year, month, day, hours, minutes, and seconds.

Dates can be compared with the before, after, and equals methods. Methods are also provided for converting a date into various formatted Strings.

```
public class Date extends Object {
// public constructors
public Date();
public Date(long date);
public Date (int year, int month, int date,int hrs,int
min);
public Date (int year, int month, int date, int hrs,
int min, int sec);
public Date (String s);
// static methods
public static long UTC(int year, int month, int date, int
hrs, int min, int sec);
public static long parse(String s);
// public instance methods
public boolean after(Date when);
public boolean before(Date when);
public Object clone();
int compareTo(Date anotherDate);
public int compareTo(Object o);
```

```
public boolean equals(Object obj);
public long getTime();
public int hashCode();
public void setTime(long time);
public String toString();
}
```

The Date class provides six constructors for creating Date objects. The default constructor creates a Date object with the current system date and time. Other constructors allow Date objects to be set to other dates and times. The access methods defined by the Date class support comparisons between dates and provide access to specific date information, including the time zone offset.

after()	Tests if this date is after the specified date.
before()	Tests if this date is before the specified date.
clone()	Return a copy of this object.
compareTo()	Compares two Dates for ordering.
compareTo()	Compares this Date to another Object.
equals()	Compares two dates for equality.
getTime()	Returns the number of milliseconds since January 1, 1970, 00:00:00 GMT represented by this Date object.
setTime()	Sets this Date object to represent a point in time that is time milliseconds after January 1, 1970 00:00:00 GMT.
toString()	Converts this Date object to a String of the form:

THE BITSET CLASS

The BitSet class is used to create objects that maintain a set of bits. The bits are maintained as a growable set. The capacity of the bit set is increased as needed. Bit sets are used to maintain a list of that indicate the state of each element of a set of conditions. Flags are boolean values that are used to represent the state of an object. The class java.util.BitSet is derived directly from Object but also implements the Cloneable interface. The definition of Bitset class can be given as below:

```
public class BitSet extends Object implements
Cloneable {
// public constructors
public BitSet();
public BitSet(int nbits);
// public instance methods
public void and(BitSet set);
public void clear(int bit);
public Object clone();
public boolean equals(Object obj);
```

```
public boolean get(int bit);
public int hashCode();
public int length();
public void or(BitSet set);
public void set(int bit);
public int size();
public String toString();
public void xor(BitSet set);
}
```

Two BitSet constructors are provided. One allows the initial capacity of a BitSet object to be specified. The other is a default constructor that initializes a BitSet to a default size.

and()	logically ANDs the BitSet with another BitSet.
clear()	clears the specified bit.
clone()	overrides the clone method in Object. It can be used to clone the bit set.
equals()	used to compare the contents of two BitSets.
get()	gets the value of a specified bit in the set.
hashCode()	overrides the hashCode method in Object and can be used to get a hash code for the instance.
length()	Returns the "logical size" of this BitSet: the index of the highest set bit in the BitSet plus one.
or()	logically ORs the BitSet with another.
set()	sets the specified bit.
size()	returns the amount of space, in bits, used to store the set.
toString()	formats the BitSet as a String.
xor()	logically XORs the BitSet with another BitSet.

THE CALENDER CLASS

Calendar is an abstract base class for converting between a Date object and a set of integer fields such as YEAR, MONTH, DAY, HOUR, and so on. Subclasses of Calendar interpret a Date according to the rules of a specific calendar system. Following are the various important methods of the class:

```
public abstract  void add(int field, int amount);
public boolean after(Object when);
public boolean before(Object when);
public void clear();
public void clear(int field);
public Object clone();
public protected  void complete();
public protected abstract  void computeFields();
```

```
public protected abstract  void computeTime();
public boolean  equals(Object obj);
public int get(int field);
public int getActualMaximum(int field);
public int getActualMinimum(int field);
public static Locale[] getAvailableLocales();
public int getFirstDayOfWeek();
public abstract  int  getGreatestMinimum(int field);
public static Calendar getInstance();
public static Calendar getInstance(Locale aLocale);
public static Calendar  getInstance(Time  one zone);
public static Calendar getInstance(Time  one zone, Locale
aLocale);
public abstract  int  getLeastMaximum(int field);
public abstract  int getMaximum(int field);
public int getMinimalDaysInFirstWeek();
public abstract  int getMinimum(int field);
public Date getTime();
public protected  long getTimeInMillis();
public Time  one getTime  one();
public int hashCode();
public protected  int internalGet(int field);
public boolean isLenient();
public boolean  isSet(int field);
public abstract  void roll(int field, boolean up);
public void roll(int field, int amount);
public void set(int field, int value);
public void set(int year, int month, int date);
public void set(int year, int month, int date, int hour, int
minute);
public void set(int year, int month, int date, int hour, int
minute, int second);
public void setFirstDayOfWeek(int value);
public void setLenient(boolean lenient);
public void setMinimalDaysInFirstWeek(int value);
public void setTime(Date date);
public  protected  void setTimeInMillis(long millis);
public  void setTime  one(Time  one value);
public String toString();
```

Given below is the definition of the important methods:

add()	Date Arithmetic function.
after()	Compares the time field records.
before()	Compares the time field records.

clear()	Clears the values of all the time fields.
clear()	Clears the value in the given time field.
clone()	Overrides Cloneable
complete()	Fills in any unset fields in the time field list.
computeFields()	Converts the current millisecond time value time to field values in fields[].
computeTime()	Converts the current field values in fields[] to the millisecond time value time.
equals()	Compares this calendar to the specified object.
get()	Gets the value for a given time field.
getFirstDayOfWeek()	Gets what the first day of the week is; e.g., Sunday in US, Monday in France.
getGreatestMinimum()	Gets the highest minimum value for the given field if varies.
getLeastMaximum()	Gets the lowest maximum value for the given field if varies.
getMaximum()	Gets the maximum value for the given time field.
getMinimalDaysInFirstWeek()	Gets what the minimal days required in the first week of the year.
getMinimum()	Gets the minimum value for the given time field.
getTime()	Gets this Calendar s current time.
getTimeInMillis()	Gets this Calendar s current time as a long.
getTime one()	Gets the time zone.
hashCode()	Returns a hash code for this calendar.
internalGet()	Gets the value for a given time field.
isLenient()	Tell whether date/time interpretation is to be lenient.
isSet()	Determines if the given time field has a value set.
roll()	Time Field Rolling function.
set()	Sets the time field with the given value.
setFirstDayOfWeek()	Sets what the first day of the week is; e.g., Sunday in US, Monday in France.
setLenient()	Specify whether or not date/time interpretation is to be lenient.
setMinimalDaysInFirstWeek()	Sets what the minimal days required in the first week of the year.

setTime()	Sets this Calendar s current time with the given Date.
setTimeInMillis()	Sets this Calendar s current time from the given long value.
setTime one()	Sets the time zone with the given time zone value.

THE DICTIONARY CLASS

Dictionary class provides the abstract functions used to store and retrieve objects by key-value associations. The class allows any object to be used as a key or value. This provides great flexibility in the design of key-based storage and retrieval classes such as Hashtable and Properties classes.

Similar to the real world dictionary, which stores the words and arrange them in order to easily reach to word's definition, Java dictionary stores keys in order and lets you reach to the value of keys. In Java dictionary, one object is used as the key to access another object. This abstraction will become clearer as you investigate the Hashtable and Properties classes. The abstract class java.util.Dictionary extends Object.

```
public class Dictionary extends Object {
// public constructors
public Dictionary();
// public instance methods
public abstract Enumeration elements();
public abstract Object get(Object key);
public abstract boolean isEmpty();
public abstract Enumeration keys();
public abstract Object put(Object key, Object value);
public abstract Object remove(Object key);
public abstract int size();
}
```

Elements are added to a Dictionary using **put** and are retrieved using **get**. Elements may be deleted with **remove**. The methods **elements** and **keys** each return an enumeration of the values and keys, respectively, stored in the Dictionary.

elements()	returns an Enumeration of all elements in a Dictionary.
get()	retrieves an object from a Dictionary based on its key.
isEmpty()	used to determine if the Dictionary is empty.
keys()	returns an Enumeration of all keys in a Dictionary.
put()	inserts a new element into the Dictionary. To retrieve an element use the get method.
remove()	removes an object from a Dictionary.
size()	returns the number of elements in the Dictionary.

THE HASHTABLE CLASS

The Hashtable class implements a hash table data structure. A hash table indexes and stores objects in a dictionary using hash codes as the objects' keys. Hash codes are integer values that identify objects. They are computed in such a manner that different objects are very likely to have different hash values and therefore different dictionary keys.

The Java Hashtable class is very similar to the Dictionary class from which it is derived. Objects are added to a hash table as key-value pairs.

```
public class Hashtable extends Dictionary {
// public constructors
public Hashtable(int initialCapacity, float loadFactor);
public Hashtable(int initialCapacity);
public Hashtable();
// public instance methods
public void clear()
public Object clone()
public boolean contains(Object value)
public boolean containsKey(Object key)
public boolean containsValue(Object value)
public Enumeration elements()
public Set entrySet()
public boolean equals(Object o)
public Object get(Object key)
public int hashCode()
public boolean isEmpty()
public Enumeration keys()
public Set keySet()
public Object put(Object key, Object value)
public void putAll(Map t)
public protected void rehash()
public Object remove(Object key)
public int size()
public String toString()
public Collection values()
// protected instance methods
protected void rehash();
}
```

The Hashtable class provides three constructors. The first constructor allows a hash table to be created with a specific initial capacity and load factor. The load factor is a float value between 0.0 and 1.0 that

identifies the percentage of hash table usage that causes the hash table to be rehashed into a larger table.

The second Hashtable constructor just specifies the table's initial capacity and ignores the load factor. The default hash table constructor does not specify either hash table parameter.

clear()	removes all elements from a Hashtable.
clone()	clones the Hashtable into a new Hashtable.
contains()	searches the Hashtable to determine if a specific value is stored.
containsKey()	searches the Hashtable to determine if a specific key occurs.
elements()	method returns an enumeration of all of the element values in the instance.
get()	retrieves the object associated with the specified key.
isEmpty()	used to determine if the Hashtable is empty.
keys()	returns an enumeration of all of the keys in the instance.
put()	inserts a new element into the Hashtable.
rehash()	rehashes the Hashtable into a larger Hashtable.
remove()	removes an object from a Hashtable.
size()	returns the number of elements in the Hashtable.
toString()	overrides the toString method in Object and formats the contents of the Hashtable as a String.

```
import java.lang.System;
import java.util.Hashtable;
import java.util.Enumeration;
public class HashApp {
public static void main(String args[]){
Hashtable h = new Hashtable();
h.put("Title","Java 2");
h.put("Author","Munishwar Gulati");
h.put("Pages","240");
h.put("ISBN","81-87870");
System.out.println("Book: "+h);
Enumeration enum = h.keys();
System.out.print("keys: ");
while (enum.hasMoreElements())
    System.out.print(enum.nextElement()+", ");
System.out.print("\nelements: ");
enum = h.elements();
while (enum.hasMoreElements())
```

```
        System.out.print(enum.nextElement()+", ");
    System.out.println();
    System.out.println("Title: "+h.get("Title"));
    System.out.println("Author: "+h.get("Author"));
    System.out.println("Pages: "+h.get("Pages"));
    System.out.println("ISBN: "+h.get("ISBN"));
    h.remove("ISBN");
    System.out.println("Book: "+h);
    }
    }
```

The output of the above program is as follows:

```
Book:     {ISBN=81-87870, Author=Munishwar Gulati,
          Title=Java 2, Pages=240}
keys:     ISBN, Author, Title, Pages, "elements: 81-87870,
          Munishwar Gulati, Java 2, 240,
Title:    Java 2
Author:   Munishwar Gulati
Pages:    240
ISBN:     81-87870
Book:     {Author=Munishwar Gulati, Title=Java 2,
          Pages=240}
```

THE PROPERTIES CLASS

The Properties class - a subclass of Hashtable, can be read from or written to a stream. This class can be used to store keys and associated values. Through its save and load methods, Properties can be written to disk. It also provides the capability to specify a set of default values to be used if a specified key is not found in the table. The default values themselves are specified as an object of class Properties. This allows an object of class Properties to have a default Properties object, which in turn has its own default properties, and so on.

```
public class Properties extends Hashtable {
// public constructors
public Properties();
public Properties(Properties defaults);
// public instance methods
public String getProperty(String key);
public String getProperty(String key, String
defaultValue);
public void list(PrintStream out);
public void list(PrintStream out);
public synchronized void load(InputStream in);
public Enumeration propertyNames();
```

```
public Object setProperty(String key, String value);
public void store(OutputStream out, String header);
// protected variables
protected Properties defaults;
}
```

GetProperty()	used to retrieve a property based on its key.
list()	reads a set of properties from the specified InputStream.
propertyNames()	returns an enumeration of all the property names in the instance.
setProperty()	Calls the hashtable method put.
store()	Writes this property list (key and element pairs) in this Properties table to the output stream in a format suitable for loading into a Properties table using the load method.

```
import java.lang.System;
import java.util.Properties;
public class SystemApp {
public static void main(String args[]) {
long time = System.currentTimeMillis();
System.out.print("Milliseconds elapsed since January 1, 1970: ");
System.out.println(time);
Properties p=System.getProperties();
p.list(System.out);
System.exit(13);
}
}
```

```
-- listing properties --
java.specification.name=Java Platform API Specification
awt.toolkit=sun.awt.windows.WToolkit
java.version=1.2.2
java.awt.graphicsenv=sun.awt.Win32GraphicsEnvironment
user.timezone=Asia/Calcutta
java.specification.version=1.2
java.vm.vendor=Sun Microsystems Inc.
user.home=C:\WINDOWS
java.vm.specification.version=1.0
os.arch=x86
java.awt.fonts=
java.vendor.url=http://java.sun.com/
```

```
user.region=US
file.encoding.pkg=sun.io
java.home=C:\JDK1.2.2\JRE
java.class.path=C:\jdk1.2.2\LIB\CLASSES.ZIP;c:\jdk1.2.2
line.separator=
java.ext.dirs=C:\JDK1.2.2\JRE\lib\ext
java.io.tmpdir=C:\WINDOWS\TEMP \
os.name=Windows 95
java.vendor=Sun Microsystems Inc.
java.awt.printerjob=sun.awt.windows.WPrinterJob
java.library.path=C:\JDK1.2.2\BIN;.; C:\WINDOWS\SYSTEM;
java.vm.specification.vendor=Sun Microsystems Inc.
sun.io.unicode.encoding=UnicodeLittle
file.encoding=Cp1252
java.specification.vendor=Sun Microsystems Inc.
user.language=en
user.name=M N Gulati
java.vendor.url.bug=http://java.sun.com/cgi-bin/bugreport...
java.vm.name=Classic VM
java.class.version=46.0
java.vm.specification.name=Java Virtual Machine Specification
sun.boot.library.path=C:\JDK1.2.2\JRE\bin
os.version=4.90
java.vm.version=1.2.2
java.vm.info=build JDK-1.2.2-W, native threads, sy...
java.compiler=symcjit
path.separator=;
file.separator=\
user.dir=C:\jdk1.2.2
sun.boot.class.path=C:\JDK1.2.2\JRE\lib\rt.jar;C:\JDK1.2....
Milliseconds elapsed since January 1, 1970: 998373250270
```

THE VECTOR CLASS

The Vector class provides the capability to implement a growable array. The array grows larger as more elements are added to it. A Vector is analogous to a linked list in other languages or class libraries. A Vector stores items of type Object, so it can be used to store instances of any Java class. A single Vector may store different elements that are instances of different classes.

As the Vector class implements a dynamically allocated list of objects, it attempts to optimize storage by increasing the storage capacity of the list when needed by increments larger than just one object. Typically, with this mechanism, there is some excess capacity in the

list. When this capacity is exhausted, the list is reallocated to add another block of objects at the end of the list.

```
public class Vector extends Object {
// public constructors
public Vector(int initialCapacity, int
capacityIncrement);
public Vector(int initialCapacity);
public Vector();
// public instance methods
public void add(int index, Object element);
public boolean add(Object o);
public boolean addAll(Collection c);
public boolean addAll(int index, Collection c);
public final synchronized void addElement(Object obj);
public final int capacity();
public void clear();
public synchronized Object clone();
public final boolean contains(Object elem);
public boolean containsAll(Collection c)
public final synchronized void copyInto(Object anArray[]);
public final synchronized Object elementAt(int index);
public final synchronized Enumeration elements();
public final synchronized void ensureCapacity(int
minCapacity);
public boolean equals(Object o)
public Object firstElement()
public Object get(int index)
public int hashCode()
public final int indexOf(Object elem);
public final synchronized int indexOf(Object elem, int
index);
public final synchronized void insertElementAt(Object
obj, int index);
public final boolean isEmpty();
public final synchronized Object lastElement();
public final int lastIndexOf(Object elem);
public final synchronized int lastIndexOf(Object elem, int
index);
public Object remove(int index)
public boolean remove(Object o)
public boolean removeAll(Collection c)
public final synchronized void removeAllElements();
public final synchronized boolean removeElement(Object
obj);
```

```
public final synchronized void removeElementAt(int index);
public final synchronized void setElementAt(Object obj, int index);
public protected void removeRange(int fromIndex, int toIndex)
public boolean retainAll(Collection c)
public Object set(int index, Object element)
public void setElementAt(Object obj, int index)
public final synchronized void setSize(int newSize);
public final int size();
public List subList(int fromIndex, int toIndex)
public Object[] toArray()
public Object[] toArray(Object[] a)
public final synchronized String toString();
public final synchronized void trimToSize();
// protected variables
protected int capacityIncrement;
protected int elementCount;
protected Object elementData[];
}
```

Following is the definition of important method.

add()	Inserts the specified element at the specified position in this Vector.
addAll()	Appends all of the elements in the specified Collection to the end of this Vector
addAll()	Inserts all of the elements in in the specified Collection into this Vector at the specified position.
addElement()	Adds the specified component to the end of this vector, increasing its size by one.
capacity()	Returns the current capacity of this vector.
clear()	Removes all of the elements from this Vector.
clone()	Returns a clone of this vector.
contains()	Tests if the specified object is a component in this vector.
containsAll()	Returns true if this Vector contains all of the elements in the specified Collection.
copyInto()	Copies the components of this vector into the specified array.
elementAt()	Returns the component at the specified index.
equals()	Compares the specified Object with this Vector for equality.
firstElement()	Returns the first component of this vector.

get()	Returns the element at the specified position in this Vector.
hashCode()	Returns the hash code value for this Vector.
indexOf()	Searches for the first occurence of the given argument, testing for equality using the equals method.
insertElementAt()	Inserts the specified object as a component in this vector at the specified index.
isEmpty()	Tests if this vector has no components.
lastElement()	Returns the last component of the vector.
lastIndexOf()	Returns the index of the last occurrence of the specified object in this vector.
lastIndexOf()	Searches backwards for the specified object, starting from the specified index, and returns an index to it.
remove()	Removes the element at the specified position in this Vector.
removeAll()	Removes from this Vector all of its elements that are contained in the specified Collection.
removeElement()	Removes the first occurrence of the argument from this vector.
removeElementAt()	Deletes the component at the specified index.
removeRange()	Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.
retainAll()	Retains only the elements in this Vector that are contained in the specified Collection.
set()	Replaces the element at the specified position in this Vector with the specified element.
setElementAt()	Sets the component at the specified index of this vector to be the specified object.
setSize()	Sets the size of this vector.
size()	Returns the number of components in this vector.
subList()	Returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive.
toArray()	Returns an array containing all of the elements in this Vector in the correct order.
toString()	Returns a string representation of this Vector, containing the String representation of each element.
trimToSize()	Trims the capacity of this vector to be the vector's current size.

```
import java.lang.System;
import java.util.Vector;
import java.util.Enumeration;
public class VectorApp {
public static void main(String args[]){
Vector v = new Vector();
v.addElement("Media");
v.addElement("Books");
v.addElement("the");
v.insertElementAt("Silicon",0);
v.insertElementAt("are",3);
v.insertElementAt("Best",5);
System.out.println("Size: "+v.size());
Enumeration enum = v.elements();
while (enum.hasMoreElements())
    System.out.print(enum.nextElement()+" ");
System.out.println();
v.removeElement("are");
v.insertElementAt("-",3);
System.out.println("Size: "+v.size());
for(int i=0;i<v.size();++i)
    System.out.print(v.elementAt(i)+" ");
System.out.println();
}
}
```

The output of above program is as follows:

```
Size: 6
Silicon Media Books are the Best
Size: 6
Silicon Media Books - the Best
```

THE OBSERVABLE CLASS

The class java.util.Observable extends Object directly. An Observable class is a class that may be watched or monitored by another class that implements the Observer interface. Associated with an Observable instance is a list of Observers. Whenever the Observable instance changes, it can notify each of its Observers. By using Observable and Observer classes, you can achieve a better partitioning of your code by decreasing the reliance of one class on another.

```
public class Observable extends Object {
// public constructors
public Observable();
```

```
// public instance methods
public synchronized void addObserver(Observer o);
public synchronized int countObservers();
public synchronized void deleteObserver(Observer o);
public synchronized void deleteObservers();
public synchronized boolean hasChanged();
public void notifyObservers();
public synchronized void notifyObservers(Object arg);
// protected instance methods
protected synchronized void clearChanged();
protected synchronized void setChanged();
}
```

addObserver() adds an Observer to the list of objects that are observing this instance.

clearChanged() clears the internal flag that indicates an Observable instance has changed.

countObservers() counts the number of Observers that are observing the instance.

deleteObserver() deletes an Observer from the list of Observers that are monitoring an Observable object.

hasChanged() used to query whether an Observable has changed.

notifyObservers() notifies all Observers that a change has occurred in the Observable object.

setChanged() sets an internal flag to indicate that an observable change has occurred within the instance.

THE STACK CLASS

The class java.util.Stack extends the class java.util.Vector, which is described later in this chapter. The Stack class in the Java library implements a Last In, First Out (LIFO) stack of objects. An item is stored on a stack by "pushing" it onto the stack. An item may subsequently be "popped" off the stack and used. The item popped off a stack will always be the most recently pushed item.

Even though they are based on the Vector class, Stack objects are typically not accessed in a direct fashion. Instead, values are pushed onto and popped off the top of the stack.

```
public class Stack extends Vector {
public Stack();                    // public constructors
// public instance methods
public empty();
public peek();
public pop();
public push(Object item);
public search(Object o);
}
```

empty()	used to determine whether the Stack contains items.
peek()	used to peek at the top item on the Stack.
pop()	retrieves the last item added to the Stack.
push()	adds a new item to the Stack.
search()	examines the Stack to see whether the specified object is in the Stack.

```
import java.lang.System;
import java.util.Stack;
public class StackApp {
public static void main(String args[]){
Stack s = new Stack();
s.push("one");
s.push("two");
s.push("three");
System.out.println("Top of stack: "+s.peek());
while (!s.empty())
System.out.println(s.pop());
}
}
```

The output of the following program is as follows:

```
Top of stack: three
three
two
one
```

THE STRINGTOKENIZER CLASS

The class java.util.StringTokenizer extends Object and implements the Enumeration interface. A StringTokenizer can be used to parse a String into a number of smaller strings called tokens. This class works specifically for what is called "delimited text," which means that each individual substring of the string is separated by a delimiter.

For example, each word in a sentence could be considered a token. However, the StringTokenizer class goes beyond the parsing of sentences. You can create a fully customized tokenizer by specifying the set of token delimiters when the StringTokenizer is created. For parsing text, the default whitespace delimiters are usually sufficient. However, you could use the set of arithmetic operators (+, *, /, and -) if parsing an expression.

```
public class StringTokenizer extends Object implements
Enumeration {
// public constructors
```

```
public StringTokenizer(String str, String delim, boolean
returnTokens);
public StringTokenizer(String str, String delim);
public StringTokenizer(String str);
// public instance methods
public int countTokens();
public boolean hasMoreElements();
public boolean hasMoreTokens();
public Object nextElement();
public String nextToken();
public String nextToken(String delim);
}
```

Because StringTokenizer implements the Enumeration interface, it includes the hasMoreElements and nextElement methods. Additionally, the methods hasMoreTokens and nextToken are provided. The hasMoreTokens method is identical to hasMoreElements, except that you may prefer the method name. The same is true of nextToken and nextElement.

countTokens() returns the number of remaining tokens.

hasMoreElements() used to determine whether the StringTokenizer contains more elements (tokens).

nextElement() overrides nextElement in the Enumeration interface and exists because StringTokenizer implements that interface.

nextToken() returns the next token in the String that is being tokenized.

THE RANDOM CLASS

The class java.util.Random is derived directly from Object.

```
public class Random extends Object {
public Random();                    // public constructors
public Random(long seed);           // public constructors
// public instance methods
public double nextDouble();
public float nextFloat();
public synchronized double nextGaussian();
public int nextInt();
public long nextLong();
public synchronized void setSeed(long seed);
}
```

nextDouble() retrieves the next number from the random number generator.

nextFloat() retrieves the next number from the random number generator.

nextGaussian()	retrieves the next value from the pseudo-random number generator.
nextInt()	retrieves the next number from the random number generator.
nextLong()	retrieves the next number from the random number generator.
setSeed()	sets a seed value for the pseudo-random number generator.

```
import java.lang.System;
import java.util.Random;
public class RandomApp {
public static void main(String args[]){
Random r = new Random();
for(int i=0;i<4;++i) {
    System.out.print(r.nextInt()+" ");
    System.out.print( \n ); }
System.out.println();
r = new Random(123456789);
for(int i=0;i<4;++i) {
    System.out.print(r.nextDouble()+" ");
    System.out.print( \n ); }
System.out.println();
r.setSeed(234567890);
for(int i=0;i<4;++i) {
    System.out.print(r.nextGaussian()+" ");
    System.out.print( \n ); }
System.out.println();}
}
```

The output of the above program is as follows :

```
1981421951
-2108552511
-1409224881
-1652793132
0.664038103272266
0.45695178590520646
0.39050647939140426
0.8933411602003871
0.11378145160284904
0.41229626309333445
-1.5726230841498485
0.07568285309772235
```

EXERCISE

1. Fill in the Blanks:
   1. Object are added in _____ as key-value pair.
   2. _____ method sets the time field with the given value.
   3. _____ method sets the time zone with the given time zone value.
   4. _____ method returns a clone of this vector.
   5. _____ method sets the size of this vector.
   6. _____ method is used to determine whether the Stack contains items.
   7. _____ method reads a set of properties from the specified InputStream.
2. Which methods are used to test whether a date is after or before a specified date.
3. What is the significance of hashtable class? How many constructors it provides?
4. How Vector class implements a dynamically allocated list of objects?
5. What is the use of Enumeration interface? Discuss a class which implements this interface.
6. Which class can be monitored by another class and which interface it uses. Discuss various methods related to this class.
7. How random is Random()? How do you generate a random integer between a and b?

I/O PACKAGE

INPUT STREAMS

OUTPUT STREAM

FILE CLASSES

I/O package

Input and output function for receiving and sending the data, are essential part of any programming language. Most programs require input from the user and in return, they output information to the screen, printer, and often to files. The Java I/O package provides an extensive set of classes that handle input and output to and from many different devices.

The Java I/O package, also known as java.io, gives classes support for reading and writing data to and from different input and output devices, including files, strings, and other data sources. The I/O package includes classes for inputting streams of data, outputting streams of data, working with files, and tokenizing streams of data. Here, we will not take an exhaustive look at every class and method contained in the I/O package. Instead, you can view this chapter as a tutorial on how to perform basic input and output using the more popular I/O classes.

Here, you will find the classification of classes on the basis of functionality of the class. Basically, classes can be input classes, output classes or File classes.

INPUT STREAMS

The Java input is similar to input stream of water and is directed in many different ways. The data in an input stream is transmitted one byte at a time. Java uses input streams as the means of reading data from an input source, such as the keyboard. The basic input stream classes supported by Java follow:

- InputStream
- BufferedInputStream
- DataInputStream
- FileInputStream

The later versions of Java also supports the charavter input stream, which are virtually identical to the input streams just listed except that they operate on characters rather than on bytes. These characters streams are called readers. The purpose of providing character-based versions of the input stream classes is to help facilitate the internationalization of character information.

- Reader
- BufferedReader
- FileReader
- StringReader

The InputStream class is an abstract class representing an input stream of bytes; all input streams are based on this class. Here is the InputStream definition:

```
public abstract class java.io.InputStream extends
java.lang.Object {
// Constructors
public InputStream();
// Methods
public int available();
public void close();
public void mark(int readlimit);
public boolean markSupported();
public abstract int read();
public int read(byte b[]);
public int read(byte b[], int off, int len);
public void reset();
public long skip(long n);
}
```

available() determines the number of bytes that can be read from the input stream without blocking. Throws **IOException** if an I/O error occurs.

close() closes the input stream, releasing any resources associated with it. Throws **IOException** if an I/O error occurs.

mark() marks the current read position in the input stream.

markSupported() determines whether the input stream supports the mark and reset methods. Throws **IOException** if an I/O error occurs.

read() reads a byte value from the input stream, blocking until the byte is read. Throws **IOException** if an I/O error occurs.

read(byte b[]) reads up to b.length bytes from the input stream into the byte array b, blocking until all bytes are read.

ead(byte b[], int off, int len)

reads up to len bytes from the input stream into the byte array b beginning off bytes into the array, blocking until all bytes are read.

reset() resets the read position in the input stream to the current mark position, as set by the mark method. Throws **IOException** if an I/O error occurs.

skip() skips n bytes of data in the input stream. Throws **IOException** if an I/O error occurs.

This class implements a buffered input stream, which allows you to read data from a stream without causing a call to the underlying system for each byte read. This is done by reading blocks of data into a buffer, where the data is readily accessible, independent of the underlying stream. Subsequent reads are much faster since they read from the buffer rather than the underlying input stream. Here is the definition for the BufferedInputStream class:

```
public class java.io.BufferedInputStream extends
java.io.FilterInputStream{
// Member Variables
protected byte buf[];
protected int count;
protected int marklimit;
protected int markpos;
protected int pos;
// Constructors
public BufferedInputStream(InputStream in);
public BufferedInputStream(InputStream in, int size);
// Methods
public int available();
public void close();
public void mark(int readlimit);
public boolean markSupported();
public int read();
public int read(byte b[], int off, int len);
public void reset();
public long skip(long n);
}
```

As you could see that the methods offered in BufferedInputStream are similar to the InputStream. However, the BufferedInputStream class does have two different constructors, which follow:

- BufferedInputStream(InputStream in)
- BufferedInputStream(InputStream in, int size)

You may note that both constructors take an InputStream object as the first parameter. The only difference between the two is the size of the internal buffer.

The DataInputStream class implements an input stream that can read Java primitive data types in a platform-independent manner.

```
public class java.io.DataInputStream extends
java.io.FilterInputStream
```

```
implements java.io.DataInput {
// Constructors
public DataInputStream(InputStream in);
// Methods
public final int read(byte b[]);
public final int read(byte b[], int off, int len);
public final boolean readBoolean();
public final byte readByte();
public final char readChar();
public final double readDouble();
public final float readFloat();
public final void readFully(byte b[]);
public final void readFully(byte b[], int off, int len);
public final int readInt();
public final long readLong();
public final short readShort();
public final int readUnsignedByte();
public final int readUnsignedShort();
public final String readUTF();
public final static String readUTF(DataInput in);
public final int skipBytes(int n);
}
```

There is only one constructor for DataInputStream, which simply takes an InputStream object as its only parameter.

Various read methods offered by DataInputStream are variations of the read() method for different fundamental data types. The type read by each method is easily identifiable by the name of the method.

ReadFully() reads up to len bytes from the data input stream into the byte array b beginning off bytes into the array, blocking until all bytes are read.

SkipBytes() skips n bytes of data in the data input stream, blocking until all bytes are skipped.

All read() methods, except **read(byte b[])** and **read(byte b[], int off, int len)**, throws **EOFException** if the end of the stream is reached before reading the value and **IOException** if an I/O error occurs.

The FileInputStream class implements an input stream for reading data from a file or file descriptor.

```
public class java.io.FileInputStream extends
java.io.InputStream
{
```

```
// Constructors
public FileInputStream(File file);
public FileInputStream(FileDescriptor fdObj);
public FileInputStream(String name);
// Methods
public int available();
public void close();
protected void finalize();
public final FileDescriptor getFD();
public int read();
public int read(byte b[]);
public int read(byte b[], int off, int len);
public long skip(long n);
}
```

This class functions exactly like the InputStream class except that it is geared toward working with files.

OUTPUT STREAM

The data going in must come out after processing. In Java, you use output streams to output data to various output devices, such as the screen. The Java output streams provide a variety of ways to output data. The primary output stream classes used in Java programming are as follows:

- OutputStream
- PrintStream
- BufferedOutputStream
- DataOutputStream
- FileOutputStream

Java also supports character output streams, which are virtually identical to the output streams just listed except that they operate on characters rather than bytes. The character output stream classes are called writers instead of output streams. A corresponding writer class implements methods similar to the output stream classes just listed except for the DataOutputStream class.

- Writer
- PrintWriter
- BufferedWriter
- FileWriter

OutputStream is an abstract class representing an output stream of bytes. All output streams are based on OutputStream. It provides a

set of methods that are the output analog to the InputStream methods.

```
public abstract class java.io.OutputStream extends
java.lang.Object {
// Constructors
public OutputStream();
// Methods
public void close();
public void flush();
public void write(byte b[]);
public void write(byte b[], int off, int len);
public abstract void write(int b);
}
```

close() closes the output stream, releasing any resources associated with it.

flush() flushes the output stream, resulting in any buffered data being written to the underlying output stream.

write(byte b[]) writes b.length bytes to the output stream from the byte array b. This method actually calls the three-parameter version of write, passing b, 0, and b.length as the parameters.

write(byte b[], int off, int len) writes len bytes to the output stream from the byte array b beginning off bytes into the array. It actually writes each byte by calling the write method that takes one parameter.

write(int b) writes a byte value to the output stream.

This class implements an output stream that has additional methods for printing basic types of data. The PrintStream class is derived from OutputStream. You can set up the stream so that it's flushed every time a newline character (\n) is written.

```
public class java.io.PrintStream extends
java.io.FilterOutputStream {
// Constructors
public PrintStream(OutputStream out);
public PrintStream(OutputStream out, boolean
autoflush);
// Methods
public boolean checkError();
public void close();
public void flush();
public void print(boolean b);
```

```
public void print(char c);
public void print(char s[]);
public void print(double d);
public void print(float f);
public void print(int i);
public void print(long l);
public void print(Object obj);
public void print(String s);
public void println();
public void println(boolean b);
public void println(char c);
public void println(char s[]);
public void println(double d);
public void println(float f);
public void println(int i);
public void println(long l);
public void println(Object obj);
public void println(String s);
public void write(byte b[], int off, int len);
public void write(int b);
}
```

checkError() flushes the underlying output stream and determines whether an error has occurred on the stream.

close() closes the print stream, releasing any resources associated with the underlying output stream.

flush() flushes the print stream, resulting in any buffered data being written to the underlying output stream.

print() prints the string representation of or different fundamental data types value to the underlying output stream.

println() prints the newline character (\n) or different fundamental data types to the underlying output stream.

write(byte b[], int off, int len)

writes len bytes to the underlying output stream from the byte array b beginning off bytes into the array.

This class implements a buffered output stream, which enables you to write data to a stream without causing a call to the underlying system for each byte written. This is done by writing blocks of data into a buffer rather than directly to the underlying output stream. The

buffer is then written to the underlying output stream when the buffer fills up or is flushed or when the stream is closed.

```
public class java.io.BufferedOutputStream extends
java.io.FilterOutputStream {
// Member Variables
protected byte buf[];
protected int count;
// Constructors
public BufferedOutputStream(OutputStream out);
public BufferedOutputStream(OutputStream out, int
size);
// Methods
public void flush();
public void write(byte b[], int off, int len);
public void write(int b);
}
```

Flush() flushes the output stream, resulting in any buffered data being written to the underlying output stream.

write(byte b[], int off, int len) writes len bytes to the buffered output stream from the byte array b beginning off bytes into the array.

write(int b) writes a byte value to the buffered output stream.

The DataOutputStream class is useful for writing primitive Java data types to an output stream in a portable way.

```
public class java.io.DataOutputStream extends
java.io.FilterOutputStream
implements java.io.DataOutput {
// Member Variables
protected int written;
// Constructors
public DataOutputStream(OutputStream out);
// Methods
public void flush();
public final int size();
public void write(byte b[], int off, int len);
public void write(int b);
public final void writeBoolean(boolean v);
public final void writeByte(int v);
public final void writeBytes(String s);
public final void writeChar(int v);
```

```
public final void writeChars(String s);
public final void writeDouble(double v);
public final void writeFloat(float v);
public final void writeInt(int v);
public final void writeLong(long v);
public final void writeShort(int v);
public final void writeUTF(String str);
}
```

There is only one constructor for DataOutputStream, which simply takes an OutputStream object as its only parameter.

Various write methods offered by DataOutputStream are variations of the write() method for different fundamental data types. The type write by each method is easily identifiable by the name of the method.

The FileOutputStream class provides a means to perform simple file output.

```
public class java.io.FileOutputStream extends
java.io.OutputStream
{
// Constructors
public FileOutputStream(File file);
public FileOutputStream(FileDescriptor fdObj);
public FileOutputStream(String name);
// Methods
public void close();
protected void finalize();
public final FileDescriptor getFD();
public void write(byte b[]);
public void write(byte b[], int off, int len);
public void write(int b);
}
```

This class functions exactly like the OutputStream class except that it is geared toward working with files.

FILE CLASSES

Java supports stream-based file input and output through the File, FileDescriptor, FileInputStream, and FileOutputStream classes. It supports direct- or random-access I/O using the File, FileDescriptor, and RandomAccessFile classes besides Random-access I/O.

The File class implements a filename in a platform-independent manner; it gives you the functionality needed to work with filenames and directories without having to deal with the complexities associated with filenames on a particular platform. The File class models an operating system directory entry, providing you with access to information about a file including file attributes and the full path where the file is located, among other things. Following are the few important methods :

```
public boolean canRead();
public boolean canWrite();
public boolean delete();
public boolean equals(Object obj);
public boolean exists();
public String getAbsolutePath();
public String getName();
public String getParent();
public String getPath();
public int hashCode();
public boolean isAbsolute();
public boolean isDirectory();
public boolean isFile();
public long lastModified();
public long length();
public String[] list();
public String[] list(FilenameFilter filter);
public boolean mkdir();
public boolean mkdirs();
public boolean renameTo(File dest);
public String toString();
}
```

CanRead() determines whether the underlying file can be read from.

CanWrite() determines whether the underlying file can be written to.

Delete() deletes the underlying file.

Equals() compares the path name of the obj File object to the path name of the underlying file.

exists() determines whether the underlying file exists by opening it for reading and then closing it.

getAbsolutePath() determines the platform-specific absolute path name of the underlying file.

getName() determines the filename of the underlying file, which doesn't include any path information.

getParent()	determines the parent directory of the underlying file, which is the immediate directory where the file is located.
getPath()	determines the path name of the underlying file.
hashCode()	calculates a hash code for the underlying file.
isAbsolute()	determines whether this object represents an absolute path name for the underlying file.
isDirectory()	determines whether the underlying file is actually a directory.
isFile()	determines whether the underlying file is a normal file-that is, not a directory.
lastModified()	determines the last modification time of the underlying file.
length()	determines the length in bytes of the underlying file.
list()	builds a list of the filenames in the directory represented by this object.
mkdir()	creates a directory based on the path name specified by this object.
mkdirs()	creates a directory based on the path name specified by this object, including all necessary parent directories.
renameTo()	renames the underlying file to the filename specified by the dest file object.
toString()	determines a string representation of the path name for the underlying file.

The RandomAccessFile class implements a random access file stream, enabling you to both read from and write to random access files. Although you can certainly use FileInputStream and FileOutputStream for file I/O, RandomAccessFile provides many more features and options.

```
public class java.io.RandomAccessFile extends
java.lang.Object
implements java.io.DataOutput java.io.DataInput {
// Constructors
public RandomAccessFile(File file, String mode);
public RandomAccessFile(String name, String mode);
// Methods
public void close();
public final FileDescriptor getFD();
public long getFilePointer();
public long length();
public int read();
```

```
public int read(byte b[]);
public int read(byte b[], int off, int len);
public final boolean readBoolean();
public final byte readByte();
public final char readChar();
public final double readDouble();
public final float readFloat();
public final void readFully(byte b[]);
public final void readFully(byte b[], int off, int len);
public final int readInt();
public final String readLine();
public final long readLong();
public final short readShort();
public final int readUnsignedByte();
public final int readUnsignedShort();
public final String readUTF();
public void seek(long pos);
publive void setLength(long newLength);
public int skipBytes(int n);
public void write(byte b[]);
public void write(byte b[], int off, int len);
public void write(int b);
public final void writeBoolean(boolean v);
public final void writeByte(int v);
public final void writeBytes(String s);
public final void writeChar(int v);
public final void writeChars(String s);
public final void writeDouble(double v);
public final void writeFloat(float v);
public final void writeInt(int v);
public final void writeLong(long v);
public final void writeShort(int v);
public final void writeUTF(String str);
}
```

close()	closes the random access file stream, releasing any resources associated with it.
getFD()	determines the file descriptor associated with the random access file stream.
getFilePointer()	determines the current read/write position in bytes of the random access file stream, which is the offset of the read/write position from the beginning of the stream.
length()	determines the length in bytes of the underlying file.

read() reads a byte value from the random access file stream, blocking until the byte is read.

Various other read methods are variations of the read() method for different fundamental data types. The type read by each method is easily identifiable by the name of the method.

readFully(byte b[], int off, int len)

reads up to len bytes from the random access file stream into the byte array b beginning off bytes into the array, blocking until all bytes are read.

readLine() reads a line of text from the random access file stream, blocking until either a newline character (\n) or a carriage return character (\r) is read.

seek() sets the current stream position to the specified absolute position.

skipBytes() skips n bytes of data in the random access file stream, blocking until all bytes are skipped.

write() writes b.length bytes to the random access file stream from the byte array b.

Various write methods are variations of the write() method for different fundamental data types. The type write by each method is easily identifiable by the name of the method.

All read() methods throws **EOFException** if the end of the stream is reached before reading the value and **IOException** if an I/O error occurs.

EXERCISE

1. How can you read data from a file? How can you write and append data to a file?
2. What is the difference between reader and input streams?
3. Fill in the Blanks:
   a) _____ method closes the random access file stream.
   b) _____ method creates a directory.
   c) _____ method flushes the print stream.
   d) _____ method prints the newline character.
   e) _____ method writes a byte value to the buffered output stream.
   f) _____ method determines the length in bytes of the underlying file.

JAVA GRAPHICS

THE COORDINATES
THE GRAPHICS CLASS
THE COLORS
THE COLOR CLASS
THE FONT CLASS
THE FONTMETRICS CLASS
THE DIMENSION CLASS
THE IMAGE CLASS

Java Graphics

The Java Abstract Windowing Toolkit (AWT) provides numerous classes that support window program development. These classes are used to draw text and graphics, create and organize windows, implement GUI components, handle events, perform image processing and obtain access to the native Windows implementation. The most important classes included in the windowing package are as follow:

- Graphical classes
- Font classes
- Dimension classes
- Components Classes
- Container Classes
- Layout manager classes
- The MediaTracker class

THE COORDINATES

All graphical computing systems use some sort of coordinate system to specify the nature of points in the system. Coordinate systems typically spell out the origin (0,0) of a graphical system, as well as the axes and directions of increasing value for each of the axes.

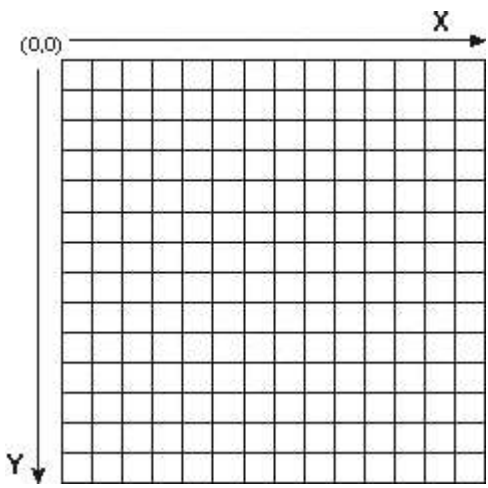


Fig 11.1 The Java Coordinate System

The Graphics class uses a standard computer coordinate system with the origin in the upper- left corner. All coordinate measurements in Java are done in pixels. The size of items, therefore, in absolute units such as inches or millimeters, differs on various machines due to differing pixels/inch values.

THE GRAPHICS CLASS

The Graphics class is part of the AWT. It's contained in java.awt.Graphics. It's the basic class for everything you'll draw on-screen. Applets have associated Graphics instances, as do various components such as buttons. Drawing methods, such as drawLine, work on a Graphics instance, everytime you draw something, you will call a Graphics instance:

```
public void paint(Graphics g) {
    g.drawLine(10,10,20,20);
    }
```

The following are few of the important methods of Graphics Class:

public abstract Graphics create()

public Graphics create(int x, int y, int width, int height)

public abstract void translate(int x, int y)

public abstract Color getColor()

public abstract void setColor(Color c)

public abstract void setPaintMode()

public abstract void setXORMode(Color c1)

public abstract Font getFont()

public abstract void setFont(Font font)

public FontMetrics getFontMetrics()

public abstract FontMetrics getFontMetrics(Font f)

public abstract void clipRect(int x, int y, int width, int height)

public abstract void copyArea(int x, int y, int width, int height, int dx, int dy)

public abstract void drawLine(int x1, int y1, int x2, int y2)

public abstract void fillRect(int x, int y, int width, int height)

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

public abstract void clearRect(int x, int y, int width, int height)

public abstract void drawRoundRect(int x, int y, int width, int height,int arcWidth, int arcHeight)

public abstract void fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)

public void draw3DRect(int x, int y, int width, int height, boolean raised)

public void fill3DRect(int x, int y, int width, int height, boolean raised)

public abstract void drawOval(int x, int y, int width, int height)

public abstract void fillOval(int x, int y, int width, int height)

public abstract void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)

public abstract void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)

public abstract void drawPolygon(int xPoints[], int yPoints[], int nPoints)

public void drawPolygon(Polygon p)

public abstract void fillPolygon(int xPoints[], int yPoints[], int nPoints)

public void fillPolygon(Polygon p)

public abstract void drawString(String str, int x, int y)

public void drawChars(char data[], int offset, int length, int x, int y)

public void drawBytes(byte data[], int offset, int length, int x, int y)

public abstract boolean drawImage(Image img, int x, int y, ImageObserver observer)

public abstract boolean drawImage(Image img, int x, int y, int width, int height, ImageObserver observer)

public abstract boolean drawImage(Image img, int x, int y, Color bgcolor, ImageObserver observer)

public abstract boolean drawImage(Image img, int x, int y, int width, int height, Color bgcolor, ImageObserver observer)

public abstract void dispose()

public void finalize()

public String toString()

Following is the description of important methods of Graphics Class.

create()	creates a new graphics object.
create(int x, int y, int width, int height)	creates a new Graphics object using the specified parameters.
translate()	translates the Graphics object to the new x and y origin coordinates.
getColor()	returns the current color.
setColor()	sets the current color.
setPaintMode()	sets the paint mode to overwrite the destination with the current color.
setXORMode()	sets the paint mode to XOR the current colors with the specified color.
getFont()	returns the current font used for the graphics context.
setFont()	sets the graphics context's font.
getFontMetrics()	returns the font metrics for the current font.
clipRect()	sets the current clipping rectangle for the Graphics class.

copyArea()	copies a specified section of the screen to another location.
drawLine()	draws a line on the graphics context from one point to another point specified by the input parameters.
fillRect()	fills the specified rectangular region with the current Color.
drawRect()	draws the outline of a rectangle using the current color and the specified dimensions.
clearRect()	clears a rectangle by filling it with the current background color of the current drawing surface.
drawRoundRect()	draws the outline of a rectangle with rounded edges using the current color and the specified coordinates.
fillRoundRect()	fills a rectangle with rounded edges using the current color and the specified coordinates.
draw3DRect()	draws a highlighted 3D rectangle at a default viewing angle.
fill3DRect()	fills a highlighted 3D rectangle using the current color and specified coordinates at a default viewing angle.
drawOval()	draws the outline of an oval shape using the current color and the specified coordinates.
fillOval()	fills an oval using the current color and the specified coordinates.
drawArc()	draws an arc outline using the current color that is bounded by the specified input coordinates.
fillArc()	fills an arc using the current color that is bounded by the specified input coordinates.
drawPolygon()	draws a polygon using the current color and the specified coordinates.
fillPolygon()	fills a polygon using the current color and the specified coordinates.
drawString()	draws a string using the current font at the specified coordinates.
drawChars()	draws a string using the current font at the specified coordinates.
drawBytes()	draws a string using the current font at the specified coordinates.
drawImage()	draws an image at a specified location within the specified bounding rectangle.
dispose()	disposes of the Graphics object.
finalize()	disposes of the Graphics object once it is no longer referenced.

toString()	returns a string representation of the Graphics object.

To draw a line in an applet, you call drawLine() in the applet's paint() method, as shown in the following example:

```
public void paint(Graphics g) {
    g.drawLine(0, 0, 15, 55);
    g.drawLine(15, 55, 30, 85);
    g.drawLine(30, 85, 10, 85);
}
```

The output of the above statements is as follows:

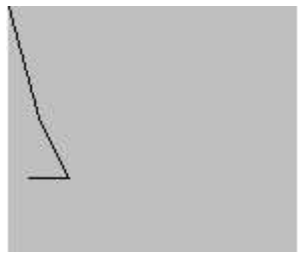


Fig 11.2 Drawing lines

To draw a rectangle using drawRect(), just call it from the paint() method like this:

```
public void paint(Graphics g) {
g.drawRect(0, 0, 30, 100);
}
```

The drawRoundRect() method requires two more parameters than drawRect(): arcWidth and arcHeight.

```
public void paint(Graphics g) {
g.drawRoundRect(0, 0, 30, 100, 20, 6);
}
```

The results of above two codes are shown in Fig

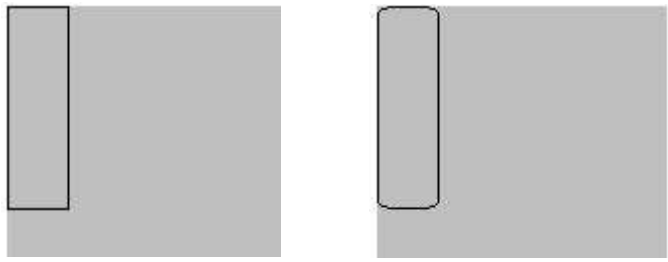


Fig 11.3 Drawing Rectangles

You can use drawLine() method to draw polygons, as we have earlier or use drawPolygon() method.

The following example uses the drawPolygon() method to draw a closed polygon:

```
public void paint(Graphics g) {
int xPnt[] = {5, 45, 80, 100, 15, 5};
int yPnt[] = {15, 50, 90, 65, 40, 15};
g.drawPolygon(xPnt, yPnt, xPnt.length);
}
```

The results of this code are shown in Fig.

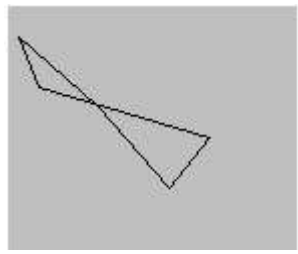


Fig 11.4 Drawing Polygon

The method for drawing ovals is drawOval(). You can use the drawOval() method as shown below:

```
public void paint(Graphics g) {
    g.drawOval(15, 20, 50, 100);
}
```

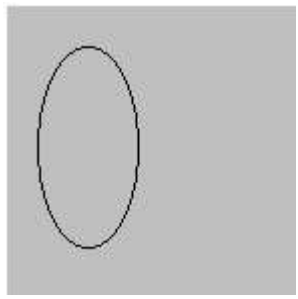


Fig 11.5 Drawing Oval

Arcs are section of Oval specified by start angle and arcangle. Following is an example of drawing arc using the drawArc() method:

```
public void paint(Graphics g) {
    g.drawArc(5, 10, 150, 75, 95, 115);
}
```

The results of this code are shown in Fig.

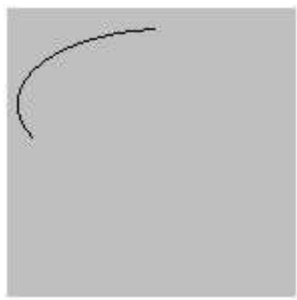


Fig 11.6 Drawing Arc

In Java, you create a font for the text and select it as the font to be used by the graphics context before actually drawing any text. The Font object models a textual font and includes the name, point size, and style of the font. The setFont() method selects a font into the current graphics context.

```
Font f = new Font("Arial", Font.BOLD + Font.ITALIC, 20);
g.setFont(f);
```

Now you're ready to draw some text using the font you've created, sized up, and selected. You use the drawString() method, defined in the Graphics class, for drawing text.

```
g.drawString("SILICONMEDIA",10,40);
```

Fig 11.7 Drawing Text

Images are rectangular graphical objects composed of colored pixels. Each pixel in an image describes the color at that particular location of the image. Java provides support for working with 32-bit images, which means that each pixel in an image is described using 32 bits

All the methods defined for drawing images, i.e. drawImage(), are variations on the same theme - they all draw an image at a certain location as defined by the parameters x and y.

The process of drawing an image involves calling the getImage() method to load the image, followed by a call to drawImage(), which actually draws the image on a graphics context.

For example, the following example explains, how easy it is to paste images on the web.

```
public void paint(Graphics g) {
Image img = getImage(getCodeBase(), "Java.bmp");
g.drawImage(img, 0, 0, this);
}
```

Fig 11.8 Drawing Image

Here, the getCodeBase() method is used to specify the applet directory in which applet resources are usually located, while the image name itself is simply given as a string.

THE COLORS

The main function of color in a computer system is to accurately reflect the physical nature of color within the confines of a graphical system.

The AWT's normal color model is RGB; in this model, a color is defined by its red, green, and blue components. Each of these three values is an int with a value between 0 and 255. An RGB value can be stored in an int with bits 31 through 24 being unused, bits 23 through 16 for the red value, bits 15 through 8 storing the green value, and bits 0 through 7 for the blue component of the color.

Following table shows the numeric values for the red, green, and blue components of some basic colors.

Color	Red	Green	Blue
White	255	255	255
Black	0	0	0
Red	255	0	0
Green	0	255	0
Blue	0	0	255
Yellow	255	255	0
Purple	255	0	255

The AWT package also provides you another color model - the HSB model. In this model, colors are represented as hue, saturation, and brightness.

Hue The value runs from 0 to 1/4 PI, with 0 corresponding to red and 1/4 PI to violet.

Saturation Determines how far from gray a color is. The value runs from 0 to 1. A value of 0 corresponds to a gray scale. By varying this parameter, you can fade out a color picture.

Brightness Determines how bright a given color is. The value runs from 0 to 1. A brightness of 0 gives you black. A value of 1 gives you the brightest color of the given hue and saturation possible.

Using HSB color model, you can convert a color image to black and white by just setting the saturation of all the pixels to 0. If you change hue from 0 to 1/4 PI, you can get a countimous spectrum of colour.

THE COLOR CLASS

The class hierarchy for the Color class derives from the class java.awt.Object. Following are the important methods of Color class:

public final static Color white
public final static Color lightGray
public final static Color gray
public final static Color darkGray
public final static Color black
public final static Color red
public final static Color pink
public final static Color orange
public final static Color yellow
public final static Color green
public final static Color magenta
public final static Color cyan
public final static Color blue
public Color(int r, int g, int b)
public Color(int rgb)
public Color(float r, float g, float b)
public int getRed()
public int getGreen()
public int getBlue()
public int getRGB()
public Color brighter()
public Color darker()
public int hashCode()
public boolean equals(Object obj)
public String toString()
public static Color getColor(String nm)
public static Color getColor(String nm, Color v)
public static Color getColor(String nm, int v)
public static int HSBtoRGB(float hue, float saturation, float brightness)
public static float[] RGBtoHSB(int r, int g, int b, float[] hsbvals)
public static Color getHSBColor(float h, float s, float b)

The public final static Color <colorname> represent the Static value representing the <colorname>. The color class also provides three constructors that allow a color to be constructed from its red, green, and blue (RGB) color components. Following are the descripion of various methods of Color class.

getRed() returns the red component of this Color.

getGreen() method returns the green component of this Color.

getBlue()	returns the blue component of this Color.
getRGB()	returns the RGB value of this Color.
brighter()	brightens this Color by approximately 1.5 times brighter than the current color by modifying the RGB color value.
darker()	Returns a color about 70 percent as bright as the original color.
hashCode()	returns this Color's hash code.
equals()	compares the Object parameter with this Color object.
toString()	returns a string representation of the Color class.
getColor()	returns the specified color property based on the name that is passed in.
HSBtoRGB()	Produces an integer containing the RGB values of a color-with blue in the lowest byte, green in the second byte, and red in the third byte-when given the three HSB values.
RGBtoHSB()	Returns an array of floats containing the h, s, and b values of the equivalent color to the specified r, g, and b values.
getHSBColor()	returns a Color object representing the RGB value of the input HSB parameters.

Consider the following program, which displays the variations of color from red to green to blue with changing hue. Here, we have taken a variation in hue and using for loop, filled the rectangle with changing hue.

```
import java.awt.*;
import java.applet.Applet;
public class colourvar extends Applet{
    int count;
    int rwidth;
public void init()
{
  }
public void paint(Graphics g)
{
    float hu,sa,br;
    int xpt,ypt,i;
    Color temp_color;
    int blockwidth,blockheight;
    float incr;
    count = 256;
    incr = (float)(0.25*Math.PI/count);
```

```
        // Hue change form 0 to 90  and we are dividing it into
        // 256 parts to fill 256 blocks
        rwidth = 1024;
        blockwidth =(int)(rwidth/count); /* width of rectangle*/
        blockheight = 150;
        sa = (float)1.0;                /* highest saturation*/
        br = (float)1.0;                /* max brightness*/
        ypt = 10;
        xpt = 0;
        for(i=0;i<count;i++) {
                hu = incr*i;
                temp_color = Color.getHSBColor(hu,sa,br);
                                        /* new colour*/
                g.setColor(temp_color);  /* setting new color*/
                xpt+= blockwidth;
        // value x changing which indicated that rectangle x point
        //is moving to right by width of rectangle
                g.fillRect(xpt ,ypt ,blockwidth,blockheight);
        /* drawing filled rectangle with new color*/
                }
    }
    }
```

The output here is printed in black and white, but on your coloured screen, you will get the continuous colours from red to green to blue.

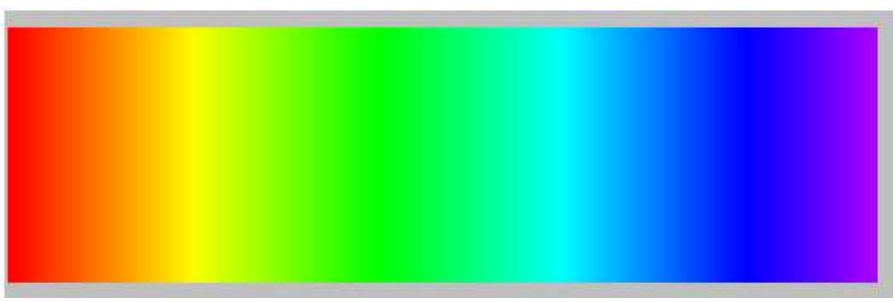


Fig 11.9 Color Rainbow

THE FONT CLASS

The class hierarchy for the Font class derives from the class java.lang.Object. The Font class implements a system-independent set of fonts that control text display. Java font names are mapped to system-supported fonts. The Courier, Dialog, DialogInput, Helvetica, TimesRoman, and ZapfDingbats fonts are the system-independent font names provided by Java.

```
public class Font {
```

```
public static final int PLAIN
public static final int BOLD
public static final int ITALIC
public Font(String name, int style, int size)
public String getFamily()
public String getName()
public int getStyle()
public int getSize()
public boolean isPlain()
public boolean isBold()
public boolean isItalic()
public static Font getFont(String nm)
public static Font getFont(String nm, Font font)
public int hashCode()
public boolean equals(Object obj)
public String toString()
}
```

getFamily()	returns the font family that this font belongs to.
getName()	returns the name of the Font object.
getStyle()	returns the style of the Font object.
getSize()	returns the size of the Font object.
isPlain()	returns the plain style state of the Font.
isBold()	returns the bold style state of the Font.
isItalic()	returns the italic style state of the Font.
getFont()	returns a Font based on the system properties list and the name passed in.
hashCode()	returns a hash code for this font.
equals()	compares an object with the Font object.
toString()	returns a string representation of the Font.

THE FONTMETRICS CLASS

You'll find that you often need to precisely position text in an applet. To precisely position the text, you must be aware of certaing basic font measurement terms:

Height()	The height of the tallest character in a font.
Baseline()	The bottom of all characters are positioned on this imaginary line.
Ascent()	Measures the height of the character above the baseline.
Descent()	Measures the height (or, more appropriately, depth) of the character below the baseline.

The FontMetrics class provides an easy way to find out how much space text drawn with a given instance of Font will be. The class hierarchy for the FontMetrics class derives from the class java.lang.Object. Following are the important methods :

```
public Font getFont()
public int getLeading()
public int getAscent()
public int getDescent()
public int getHeight()
public int getMaxAscent()
public int getMaxDescent()
public int getMaxDescent()
public int getMaxAdvance()
public int charWidth(int ch)
public int charWidth(char ch)
public int stringWidth(String str)
public int charsWidth(char data[], int off, int len)
public int bytesWidth(byte data[], int off, int len)
public int[] getWidths()
public String toString()
```

getFont()	returns the font that FontMetrics refers to.
getLeading()	gets the line spacing of the font.
getAscent()	gets the ascent value for a Font.
getDescent()	gets the descent value for a Font.
getHeight()	gets the height of a line of text using the current Font.
getMaxAscent()	returns the maximum value of a font's ascent.
getMaxDescent()	returns the maximum value of a font's descent.
getMaxDecent()	calls the getMaxDescent() method.
getMaxAdvance()	gets the maximum amount for a character's advance value.
charWidth()	returns the width of a particular character for the current font.
stringWidth()	returns the width of a specified string using the current font.
charsWidth()	returns the width of a specified string of characters using the current font.
bytesWidth()	returns the width of a specified array of bytes.
getWidths()	gets the advance widths of the first 256 characters of the font.

Consider the following example, which uses various Fontmetrics class methods.

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

```
public class fontm3 extends java.applet.Applet{
Font f1=new Font("Arial",Font.BOLD,20);
String message;
FontMetrics f1_met;
FontMetrics a_font;
int m_width, f1_height,f1_ascent, f1_descent;
int y;
public void paint(Graphics g){
    g.setFont(f1);
    g.setColor(Color.blue);
    f1_met = g.getFontMetrics();
    message  = "SILICONMEDIA";
    m_width = f1_met.stringWidth(message);
    f1_height = f1_met.getHeight();
    f1_ascent = f1_met.getAscent();
    f1_descent = f1_met.getDescent();
    y = 30;
    g.drawString(message,10,y);
    y += f1_height + 1;
    g.drawString("Width of message"+ m_width,10,y);
    y += f1_height + 1;
    g.drawString("Height of font"+ f1_height,10,y);
    y += f1_height + 1;
    g.drawString("Ascent of message"+ f1_ascent,10,y);
    y += f1_height + 1;
    g.drawString("Descentof message"+ f1_descent,10,y);
}
}
```

Fig 11.10 Font properties

THE DIMENSION CLASS

The Dimension class is used to represent the width and height of a two-dimensional object. It provides three constructors: a default

parameterless constructor, a constructor that creates a Dimension object using another Dimension object, and a constructor that takes width and height parameters. The class hierarchy for the Dimension class derives from the class java.lang.Object.

```
public class Dimension {
boolean equals(Object obj);
double getHeight();
Dimension getSize();
double getWidth();
int hashCode();
void setSize(Dimension d);
void setSize(double width, double height);
void setSize(int width, int height);
String toString();
}
```

THE IMAGE CLASS

The class hierarchy for the Image class derives from the class java.lang.Object. An Image class is actually an abstract class. You must provide a platform-specific implementation to use it.

```
public abstract class Image {
public abstract int getWidth(ImageObserver observer);
public abstract int getHeight(ImageObserver observer);
public abstract ImageProducer getSource();
public abstract Graphics getGraphics();
public abstract Object getProperty(String name,
ImageObserver observer);
Image getScaledInstance(int width, int height, int hints);
public static final Object UndefinedProperty;
public abstract void flush();
}
```

getWidth()	returns the width of the Image.
GetHeight()	returns the height of the Image.
GetSource()	returns the ImageProducer interface responsible for producing the Image's pixels.

EXERCISE

1. State True or False.
   1. Triangles can be drawn by using drawTriangle() method of Graphics class.
   2. Rectangles can also be drawn by using drawPolygon() method.
   3. In Java2D, the angle covered by an arc ranges from 0 to 359 degrees in clockwise direction.
   4. Ascent is the distance between the baseline and the lower limit of characters like g.
   5. Java supports only GIF format for images.
   6. Do you need to recompile your program if the URL of a program changes?
   7. An image is scaled down if the dimensions specified are larger than the size of the image.
2. What method is used to change the current color before you draw something in a program?
   (a) shiftColor()
   (b) setColor()
   (c) getColor()
3. If you want to use the height and width of an applet window to determine how big something should be drawn, what can you use?
   (a) setHeight() and SetWidth()
   (b) getHeight() and getWidth()
   (c) size().height and size().width
4. Ovals and circles don't have corners. What are the (x,y) coordinates specified with the fillOval() and drawOval() method?
5. Fill in the Blanks:
   1. _____ method returns the italic style state of the Font.
   2. _____ method measures the height of the character above the baseline
   3. _____ method returns the width of a specified array of bytes.
   4. _____ method returns the width of the Image.
   5. _____ class can be used to find information about the current font.
   6. If getImage() can't find an image, it returns a _____.
   7. _____ class supplies the methods for displaying image.

INTERACTIVE INTERFACE ELEMENTS

THE COMPONENT CLASSES
THE CONTAINER CLASS
THE CANVAS CLASS
LABEL
TEXTCOMPONENT
BUTTON
CHECKBOXES
RADIO BUTTONS
CHOICE MENU
SCROLLING LIST
FRAMES
PANELS
LAYOUT MANAGER
EVENT HANDLING

Interactive Interface Elements

These items enable the user to dynamically interact with the program. They include buttons, text display areas, drop down menus, list, text area etc. Although different operating systems tend to use slightly different interaction elements, the AWT provides a rich enough set for all platforms .

This chapter covers classes related to interactive interface elements. The Components and Containers classes introduces the GUI components supported by the AWT and the Windows classes that contain these components. The Menu related classes describes the classes that are used to implement menu bars and pull-down menus. The Layout Manager Classes are used to organize windows and lay out the components they contain. The Event handling classes describes the process of Java event handling.

THE COMPONENT CLASSES

The Component class is the superclass of the set of AWT classes that implement graphical user interface controls. These components include windows, dialog boxes, buttons, labels, text fields, and other common GUI components. The Component class provides a common set of methods that are used by all these subclasses. The class hierarchy for the Component class derives from the class java.lang.Object. Following are the various important methods of Component Class:

```
public Container getParent()
public Toolkit getToolkit()
public boolean isValid()
public boolean isVisible()
public boolean isShowing()
public boolean isEnabled()
public Color getForeground()
public synchronized void setForeground(Color  )
public Color getBackground()
public synchronized void setBackground(Color  )
public Font getFont()
public synchronized void setFont(Font  )
public synchronized ColorModel getColorModel()
public void setsize(int        , int        )
public void setsize(Dimension  )
public synchronized void setbounds(int  , int  , int
      , int        )
public void validate()
public void invalidate()
public Graphics getGraphics()
```

```
public FontMetrics getFontMetrics(Font      )
public void paint(Graphics  )
public void update(Graphics  )
public void paintAll(Graphics  )
public void repaint()
public void repaint(long    )
public void repaint(int  , int  , int       , int       )
public void repaint(long    , int  , int  , int       , int
      )
public void print(Graphics  )
public void printAll(Graphics  )
public boolean imageUpdate(Image     , int       ,
int  , int  , int  , int  )
public Image createImage(ImageProducer          )
public Image createImage(int       , int
public boolean prepareImage(Image       ,
ImageObserver          )
public boolean prepareImage(Image       , int       ,
int        , ImageObserver          )
public int checkImage(Image       , ImageObserver
         )
public int checkImage(Image       , int       , int
      , ImageObserver          )
public void addNotify()
public synchronized void removeNotify()
public void requestFocus()
public String toString()
public void list()
public void list(PrintStream     )
public void list(PrintStream     , int       )
```

This class contains more than 100 methods, few of which are explained below:

getParent()	returns this component's parent (a Container class).
GetPeer()	returns this component's peer (a ComponentPeer interface).
getToolkit()	returns the toolkit of this component. The toolkit creates the peer for the component.
isValid()	determines whether this component is valid.
isVisible()	determines whether this component is visible.
isShowing()	determines whether this component is shown on the screen.
isEnabled()	determines whether this component is currently enabled.

getlocation()	returns the location of this component in its parent's coordinate space.
size()	returns the current size of the component.
bounds()	returns the bounding rectangle of the component.
enable()	enables a component.
disable()	disables a component.
show()	shows the component.
hide()	hides the component from view.
getForeground()	returns the foreground color of the component.
setForeground()	sets the foreground color of the component.
getBackground()	returns the background color of the component.
setBackground()	sets the background color of the component.
getFont()	returns the font of the component.
setFont()	sets the font of the component.
getColorModel()	gets the color model that displays this component on an output device.
move()	moves a component to a new location within its parent's coordinate space.
setsize()	resizes the component to the specified width and height.
setBounds()	completely changes the bounding box of the component by changing its size and location.
preferredSize()	returns the preferred size of the component.
getminimumSize()	returns the minimum size of the component.
layout()	called when the component needs to be laid out.
validate()	validates a component by calling its layout() method.
invalidate()	invalidates a component, forcing the component and all parents above it to be laid out.
getGraphics()	returns a Graphics context for the component.
getFontMetrics()	returns the current FontMetrics for a specified font.
paint()	paints the component on the screen using the Graphics context parameter.
update()	repaints the component in response to a call to the repaint() method.
paintAll()	paints the component along with all its subcomponents.
repaint()	forces a component to repaint itself.
print()	prints the component using the Graphics context.
printAll()	prints the component and all of its subcomponents using the Graphics context.

imageUpdate()	repaints the component when the specified image has changed.
createImage()	creates an Image using the specified ImageProducer.
prepareImage()	prepares an Image for rendering on this component.
checkImage()	checks the status of the construction of the Image to be rendered.
inside()	determines whether the x and y coordinates are within the bounding rectangle of the component.
getComponentAt()	returns the Component at the specified x and y coordinates.
deliverEvent()	delivers an event to the component or one of its subcomponents.
postEvent()	posts an event to the component resulting in a call to handleEvent().
handleEvent()	handles individual events by the component.
mouseDown()	called if the mouse is down.
mouseDrag()	called if the mouse is dragged.
mouseUp()	called when the mouse button is released.
mouseMove()	called if the mouse is moved.
mouseEnter()	called if the mouse enters the component.
mouseExit()	called if the mouse exits the component.
keyDown()	called when a key is pressed.
keyUp()	called when a key is released.
addNotify()	notifies a component to create a peer object.
removeNotify()	notifies a component to destroy the peer object.
gotFocus()	called when the component receives the input focus.
processFocusEvent()	called when the component loses the input focus.
requestFocus()	requests the current input focus.
nextFocus()	switches the focus to the next component.
toString()	returns a string representation of the Component class.
list()	prints a listing of the component to the print stream.

THE CONTAINER CLASS

The AWT containers contain classes that can contain other elements. Windows, panels, dialog boxes, frames, and applets are all containers. Whenever you want to display a component such as a button or pop-up menu, you'll use a container to hold it.

The Container class has a number of methods that make it easy to add and remove components as well as to control the relative positioning and layout of those components.

```
public abstract class Container extends Component {
public synchronized Component getComponent(int n)
throws ArrayIndexOutOfBoundsException
public synchronized Component[] getComponents()
public Component add(Component comp)
public synchronized Component add(Component comp,
int pos)
public synchronized Component add(String name,
Component comp)
public synchronized void remove(Component comp)
public synchronized void removeAll()
public LayoutManager getLayout()
public void setLayout(LayoutManager mgr)
public synchronized void validate()
public void paintComponents(Graphics g)
public void printComponents(Graphics g)
public synchronized void addNotify()
public synchronized void removeNotify()
public void list(PrintStream out, int indent)
}
```

getComponent() returns the component at the specified index.

getComponents() returns an array of Component objects contained within the Container.

add() adds a Component to the container at the end of the container's array of components.

remove() removes the specified component from the Container's list.

removeAll() removes all components within Container.

getLayout() returns this Container's LayoutManager.

setLayout() sets the current LayoutManager of the Container.

preferredSize() returns the preferred size of this Container.

minimumSize() returns the minimum size of this Container.

paintComponents() paints each of the components within the container.

printComponents() prints each of the components within the container.

addNotify() notifies the container to create a peer interface.

removeNotify() notifies the container to remove its peer.

list() prints a list for each component within the container to the specified output stream at the specified indentation.

THE CANVAS CLASS

A canvas is an empty space-a starting point for complex interface elements such as a picture button. A canvas is a place to draw. You use it instead of just drawing to a panel, as in an applet, so you can take advantage of the Layout Manager's capability to keep your interface machine- independent.

The Canvas class implements a GUI object that supports drawing. Drawing is not implemented on the canvas itself, but on the Graphics object provided by the canvas. The Canvas class is usually subclassed to implement a custom graphics object. It provides a single, parameterless constructor and one useful method-the paint() method, which specifies how its Graphics object is to be updated.

The addNotify() method of sets the peer of the Canvas using the function getToolkit().createCanvas(). Using peer interfaces allows you to change the user interface of the canvas without changing its functionality.

LABEL

Labels are text items that appear as static text on the screen. They are similar to drawString, but are often used to identify text fields. By using a label instead of drawString, you can use the Layout Managers to control text placement in a platform- and monitor-independent manner. Following are the various constructors and methods

new Label(String label)	Produces a label with the specified string.
new Label(String label,int pos)	Produces a label with the string aligned according to the second value, which should be one of the three constants Label.CENTER, Label.LEFT, or Label.RIGHT.
String getText()	Returns the label string.
setText(String newlabel)	Changes the label text.

Consider the following applet code which adds three labels to the applet:

```
import java.awt.*;
import java.applet.Applet;
public class label extends Applet{
    Label Head;
    Label labels[];
    public void init(){
    int i;
    labels = new Label[2];
    Head= new Label("ORDER FORM", Label.CENTER);
```

```
Head.setBackground(Color.blue);
Head.setForeground(Color.white);
Font f1= new Font("Arial",Font.BOLD,20);
Head.setFont(f1);
add(Head);
labels[0] = new Label("Name". Label.LEFT );
f1= new Font("Arial",Font.BOLD,15);
labels[0].setFont(f1);
labels[1] = new Label("Address", Label.LEFT);
labels[1].setFont(f1);
        for(i=0;i<2;i++) {
        add(labels[i]);
        }
}
}
```

Fig 12.1 Using Label

Note the usage of GridLayout. Due to GridLayout, the three labels have appeared in three lines. We will be discussing about it in detail in coming topics.

TEXTCOMPONENT

This is abstract class, but it's extended by both TextFields and TextAreas. The methods for TextComponent class are also available in both those GUI elements. TextComponent provides the basic tools for finding out what text is in a Text item (getText), setting the text in an item (setText), and selecting pieces of text (setSelect).

The TextField class implements a one-line text entry field. It provides four constructors that are used to specify the width of the text field in character columns and the default text to be displayed within the field.

String getSelectedText()	Returns the text currently selected in the text item.
int getSelectionEnd()	Returns the index of the last character in the selection +1.
int getSelectionStart()	Returns the index of the first character in the current selection or the location of the insertion point if nothing is selected.
String getText()	Returns all the text in the text item.
select(int start, int stop)	Selects the text specified by input arguments.
selectAll	Selects all the text in the text item.
setEditable(boolean state)	Enables you to toggle between whether or not a text item is editable by the user.
setText(String new_text)	Enables you to set the text in the text item.

Consider the following applet code which adds text field along with labels to the applet. This code is an extension of previous code:

```
import java.awt.*;
import java.applet.Applet;
public class label7 extends Applet{
    Label Head;
    Label Head1;
    Label labels[];
    GridLayout g1;
    TextField text[];
    public void init(){
    int i;
    g1 = new GridLayout(4,2);
    setLayout(g1);
    labels = new Label[2];
    text = new TextField[2];
    Head= new Label("ORDER", Label.CENTER);
    Head1= new Label("FORM", Label.CENTER);
    Head.setBackground(Color.blue);
    Head.setForeground(Color.white);
    Font f1= new Font("Arial",Font.BOLD,20);
    Head.setFont(f1);
    Head1.setBackground(Color.blue);
    Head1.setForeground(Color.white);
    Head1.setFont(f1);
    add(Head);
```

```
        add(Head1);
        labels[0] = new Label("Name", Label.LEFT );
        text[0] = new TextField(35);
        f1= new Font("Arial",Font.BOLD,15);
        labels[0].setFont(f1);
        labels[1] = new Label("Address", Label.LEFT);
        text[1] = new TextField("Enter Your Address",30);
        labels[1].setFont(f1);
                for(i=0;i<2;i++) {
                        add(labels[i]);
                        add(text[i]);
                        }
        }
        }
```

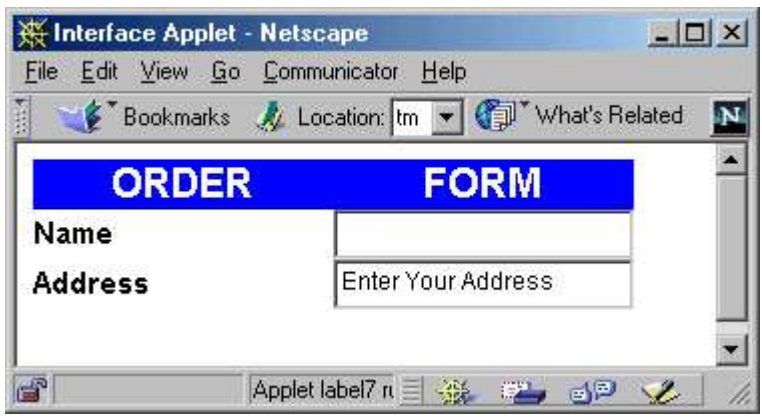


Fig 12.2 Using TextField

The TextArea class implements scrollable text entry objects that span multiple lines and columns. It provides four constructors that allow the number of rows and columns and the default text display to be specified.

new TextArea() Defines a default empty TextArea.

new TextArea(int rows, int columns)
Defines an empty TextArea with the specified number of rows and columns.

new TextArea(String the_contents)
Defines a TextArea that contains the specified string.

new TextArea(String the_contents, int rows, int columns)

Defines a TextArea containing the specified string and with a set number of rows and columns.

appendText(String new_text)

Appends the specified string to the current contents of the TextArea.

int, getColumns() Returns the current width of the TextArea in columns.

int, getRows() Returns the current number of rows in a TextArea.

insertText(String the_text, int where_to_add)

Inserts the specified string at the specified location.

replaceText(String new_text, int start, int stop)

Takes the text between start and stop, inclusive, and replaces it with the specified string.

Consider the following code for creating text area.

```
import java.awt.*;
import java.applet.Applet;
public class text1 extends Applet{
    Label Head;
    GridLayout g1;
    TextArea texta;
    public void init(){
            int i;
            g1 = new GridLayout(1,2);
            setLayout(g1);
            Head= new Label("Enter your Description");
            Font f1= new Font("Arial",Font.PLAIN,15);
            Head.setFont(f1);
            add(Head);
            texta = new TextArea(5, 20);
            add(texta)
    }
    }
```

The output is shown in Fig . 12.3

BUTTON

Java buttons are just like the buttons in every other GUI. They are text surrounded by a shape, and they generate an ACTION_EVENT event-the argument is a button's label-after the user clicks them.

Fig 12.3 Using Text Area

The class provides two Button constructors. The first constructor takes no parameters and creates a button with a blank label. The second constructor accepts a String object that is displayed as the button's label. The Button class provides methods for getting and setting its label.

addNotify() sets the peer of the button using the function getToolkit().createButton().

getLabel() returns the button's label string.

setLabel() modifies the button's label string.

```
import java.awt.*;
import java.applet.Applet;
public class buttons extends Applet{
Button a_button;
public void init()
{
a_button = new Button("SILICONMEDIA");
add(a_button);
}
}
```

Fig 12.4 Using Buttons

CHECKBOXES

A checkbox is simply an empty box and contain a checkmark when activated. They're generally used when you want the user to be able to set several options prior to making a decision. You usually don't do anything when a checkbox is checked or unchecked, you usually just read the values of the checkboxes when some other control, such as a button or menu item, is activated.

The Checkbox class provides three constructors. The first constructor takes no parameters and implements a blank checkbox. The second constructor takes a String parameter that is used as the title of the checkbox. The third constructor allows a CheckboxGroup object and the initial state of the radio button to be specified in addition to its label. It provides the following methods.

setLabel(String the_new_label)	Changes the label of a checkbox.
String getLabel()	Returns the current label as a string.
boolean getState()	Gets the current checkbox state (checked = TRUE).
setState(boolean new_state)	Sets the checkbox state.

Consider the following code:

```
import java.awt.*;
import java.applet.Applet;
public class checkboxes extends Applet{
public void init(){
    Checkbox b_1, b_2, b_3;
    box_1 = new Checkbox();
    box_2 = new Checkbox("Purchased Books");
    box_3 = new Checkbox("Checked ", null,  true);
    add(b_1);
    add(b_2);
    add(b_3);
}
}
```

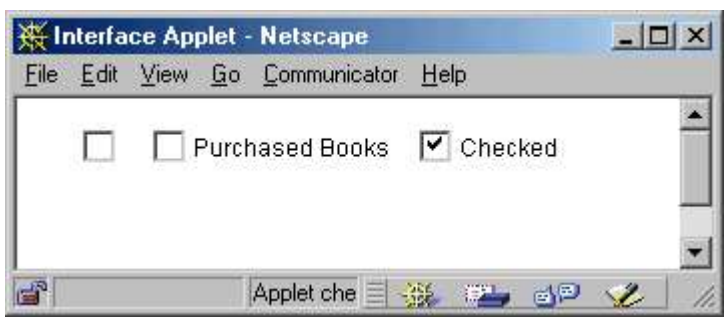


Fig 12.5 Using Checkboxes

RADIO BUTTONS

Radio buttons are created using the Checkbox class. AWT creates a radio button group by associating a CheckboxGroup instance with all the checkboxes in the group. A CheckboxGroup can be assigned with the Checkbox constructor or using the setCheckboxGroup() method. Only one object in the checkbox group is allowed to be set at any given time.

Radio buttons have only one creator method:

new Checkbox(String the_label, CheckboxGroup a_group, boolean checked?);

This creates a new Checkbox that is labeled and checked. In order to use radio buttons, you also need to create a new checkbox group Using the following code :

new CheckboxGroup();

Consider the following code:

```
import java.awt.*;
import java.applet.Applet;
public class radio  extends Applet {
public void init() {
    Label Head;
    CheckboxGroup  chkgroup;
    Checkbox b_1,b_2,b_3, b_4;
    Head= new Label("No. of CD to be purchased :");
    Font f1= new Font("Arial",Font.BOLD, 12);
    Head.setFont(f1);
    add(Head);
    chkgroup = new CheckboxGroup();
    b_1 = new Checkbox("1 CD", group, true);
    b_2 = new Checkbox("5 CD", group, false);
    b_3 = new Checkbox("10 CD", group, false);
    b_4 = new Checkbox("100 CD", group, false);
    add(b_1);
    add(b_2);
    add(b_3);
}
}
```

The Output is shown in Fig. 12.6

CHOICE MENU

Choice menus-often called pop-up menus or drop-down menu -are designed to allow the user to select an option from a menu and see the value chosen at all times. It serves a similar purpose to a group of check boxes, except that only one of the selections is visible unless the drop-down list is being displayed.

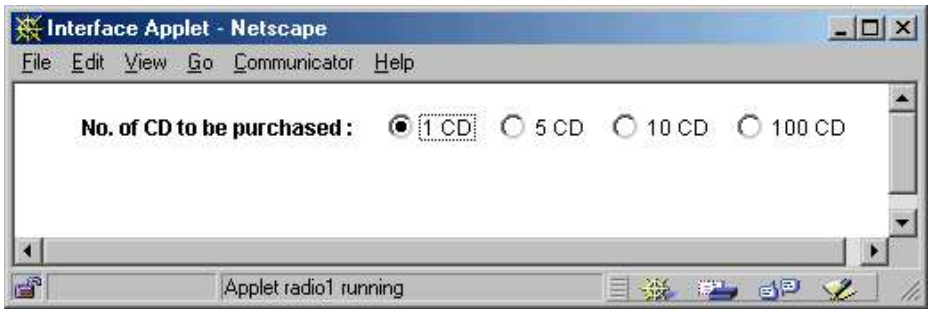


Fig 12.6 Using Radio Button

When a choice selection is made, an ACTION_EVENT is generated, and the program is then able to respond to that selection. Following are the various constructor and methods.

new Choice()	Creates a new Choice item.
addItem(String name)	Adds an item to the Choice menu. It can throw a NullPointerException. This is a synchronized method.
String getItem(int number)	Returns the text of the specified menu item (item 0 is the first item in the menu).
int getSelectIndex()	Returns the index of the currently selected item (item 0 is the first item in the menu).
String getSelectedItem()	Returns the text of currently selected menu items.
select(int item)	Changes the selection to the specified item. This is a synchronized method, and it can throw illegallArgumentException.
select(String name)	Selects the menu item for which the name is the specified string.

Consider the following listing which develops the drop-down menu displaying the name of various countries:

```
import java.awt.*;
import java.applet.Applet;
public class popup extends Applet{
public void init(){
Choice menu1;
menu1 = new Choice();
menu1.addItem("India");
menu1.addItem("Argentina");
menu1.addItem("Australia");
menu1.addItem("Austria");
```

```
menu1.addItem("Bulgaria");
menu1.addItem("France");
menu1.addItem("Germany");
menu1.addItem("Sweden");
menu1.addItem("United Kingdom");
menu1.addItem("United States");
add(a_menu);
}
}
```

Fig 12.7 Using Choice Menu

SCROLLING LIST

Scrolling lists display multiple lines of text, and each line corresponds to a selection item. Scroll bars are displayed if the text is larger than the available space. The user can select one or more of the lines. Your program can read the user's selections. It is different from Choicelist or drop-down menu, because it provides the capability to support multiple menu selections, to specify the size of the list window, and to dynamically update the list during program execution.

Lists generate three event types:

ACTION_EVENT When a list item is double-clicked. The argument is the name of the list item.

LIST_SELECT When a list item is selected. The argument is the name of the list item selected.

LIST_DESELECT When a list item is deselected. The argument is the name of the item deselected.

The list class has following methods:

add(String label)	Adds the specified item to the end of the current list of items in the list.
add(String label, int loc)	Adds the specified item to the list at the specified location.
int clear()	Removes all the entries in the list. This is a synchronized method.
int countItems()	Returns the number of items currently in the list.
del(int location)	Deletes the list item at the specified location. This is a synchronized method.
del(int first, int last)	Deletes all the items between the first and last location, inclusive.
deselect(int location)	Deselects the item at the specified location.
String get(int location)	Returns the label of the list item at the specified location.
int getRows()	Returns the number of rows currently visible to the user.
int getSelectedIndex()	Throws an ArrayIndexOutofBoundsException if it's invoked on a list where more than one item is selected.
int[] getSelectedIndexes()	Returns an array of the locations of the selected items.
String getSelectedItem()	Returns the location of the currently selected item.
String[] getSelectedItems()	Returns an array of Strings containing the names of the currently selected list items.
makeVisible(int location)	Forces the item at the specified location to be visible.
replace(String label, int loc)	Changes the name of the label specified by the value of location.
select(int location)	Selects the specified item.

Consider the following source code:

```
import java.awt.*;
import java.applet.Applet;
public class list extends Applet{
public void init(){
    List mult, single;
    mult= new List(6,true);
    single= new List();
    mult.add("ITC",6);
    mult.add("FoxPro",8);
```

```
mult.add("Java",10);
mult.add("Word",12);
mult.add("Excel");
mult.add("Flash");
single.add("Uttranchal");
single.add("Haryana");
single.add("Chandigarh");
single.add("Punjab");
single.add("Rajasthan");
single.add("Himachal");
single.add("Gujrat");
add(mult);
add(single);
}
}
```

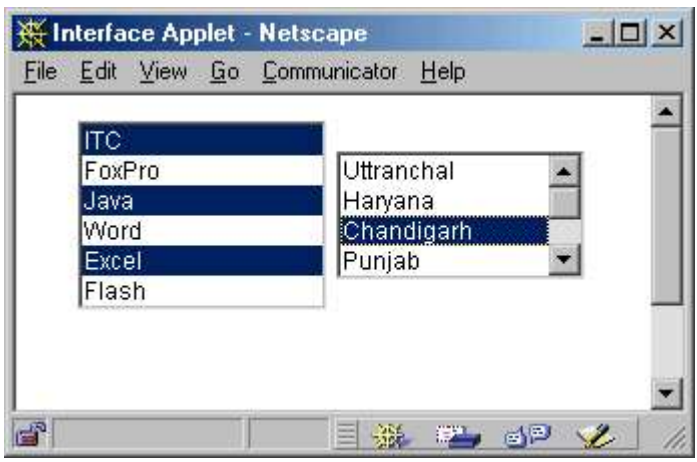


Fig 12.8 Using Scroll List

FRAMES

A Frame is a top-level window with a title and a border. The size of the frame includes any area designated for the border. Frames generate the same events as windows, which they extend: WINDOW_DESTROY, WINDOW_ICONIFY, WINDOW_DEICONIFY, and WINDOW_MOVED. The only parameter you can pass to the Frame constructor is a String, which will be the window title.

Following are the various important methods associated with Frame Class.

addNotify()	Makes this Frame displayable by connecting it to a native screen resource.
getFrames()	Returns an array containing all Frames created by the application.

getIconImage()	Gets the image to be displayed in the minimized icon for this frame.
getMenuBar()	Gets the menu bar for this frame.
getState()	Gets the state of this frame.
getTitle()	Gets the title of the frame.
isResizable()	Indicates whether this frame is resizable by the user.
paramString()	Returns the parameter String of this Frame.
remove(MenuComponent m)	Removes the specified menu bar from this frame.
removeNotify()	Makes this Frame undisplayable by removing its connection to its native screen resource.
setIconImage(Image image)	Sets the image to displayed in the minimized icon for this frame.
setMenuBar(MenuBar mb)	Sets the menu bar for this frame to the specified menu bar.
setResizable(boolean resizable)	Sets whether this frame is resizable by the user.
setState(int state)	Sets the state of this frame.
setTitle(String title)	Sets the title for this frame to the specified string.

PANELS

A panel is a container which defines an area in which the components or even the other panels can be attached. It extends from the Object.Component.Container class. Panels enable you to group items in a display in a way that might not be allowed by the available Layout Managers. The default layout manager for a panel is the FlowLayout. It defines two constructors. The constructor without any parameter creates a new panel using the default layout manager. The second constructor creates a new panel with the specified layout manager. Following are the various methods:

void addNotify()	Creates the Panel's peer.
getAccessibleContext()	Gets the acessibleContext associated with this Panel.

LAYOUT MANAGER

The AWT not only lets you specify the absolute location of components, but also gives you Layout Managers that let you define the relative placement of components that will look the same on a wide spectrum of display devices.

The organization of any object that is a subclass of the Container class is governed by a layout. The layout determines how objects of

class Component are positioned when they are added via the add() method to the Container object. Java provides five type of layouts:

- BorderLayout
- CardLayout
- FlowLayout
- GridLayout
- GridBagLayout

Other, custom layouts can also be defined.

The BorderLayout divides the container into five parts; four form the four borders of the container and the fifth is the center. You can add one component to each of these five areas. Because the component can be a panel, you can add more than one interface element, such as a button, to each of the five areas. The following are the various important methods

addLayoutComponent()	Adds the specified component to the layout, using the specified constraint object.
getHgap()	Returns the horizontal gap between components.
getLayoutAlignmentX()	Returns the alignment along the x axis.
getLayoutAlignmentY()	Returns the alignment along the y axis.
getVgap()	Returns the vertical gap between components.
invalidateLayout()	Invalidates the layout, indicating that if the layout manager has cached information it should be discarded.
layoutContainer()	Lays out the container argument using this border layout.
maximumLayoutSize()	Returns the maximum dimensions for this layout given the components in the specified target container.
minimumLayoutSize()	Determines the minimum size of the target container using this layout manager.
preferredLayoutSize()	Determines the preferred size of the target container using this layout manager, based on the components in the container.
removeLayoutComponent()	Removes the specified component from this border layout.

setHgap()	Sets the horizontal gap between components.
setVgap()	Sets the vertical gap between components.

Consider the following example:

```
import java.awt.*;
import java.applet.Applet;
public class buttonDir extends Applet {
public void init() {
setLayout(new BorderLayout());
add(new Button("TOP"), BorderLayout.NORTH);
add(new Button("BOTTOM"), BorderLayout.SOUTH);
add(new Button("LEFT"), BorderLayout.EAST);
add(new Button("RIGHT"), BorderLayout.WEST);
add(new Button("CENTER"), BorderLayout.CENTER);
}
}
```

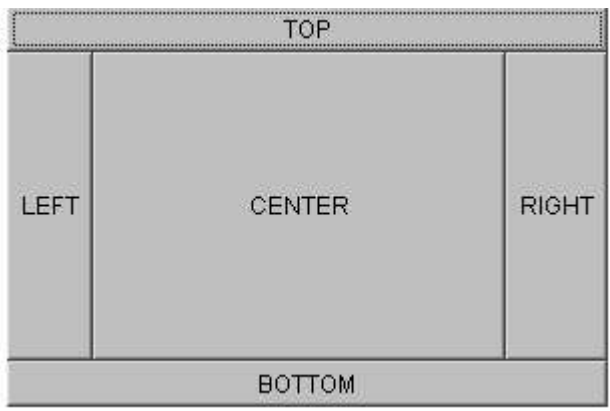


Fig 12.9 Border Layout

The CardLayout is different from the others because it enables you to create virtual screen real estate by defining multiple Cards, one of which is visible at any time. Each Card contains a panel that can contain any number of interface elements, including other panels. Following are the methods other than explained in previous Layout class.

first(Container parent)	Flips to the first card of the container.
last(Container parent)	Flips to the last card of the container.

next(Container parent)	Flips to the next card of the specified container.
previous(Container parent)	Flips to the previous card of the specified container.
show(Container parent, String name)	Flips to the component that was added to this layout with the specified name, using addLayoutComponent.

This is the default Layout Manager that every panel uses unless you use the setLayout method to change it. It keeps adding components to the right of the preceding one until it runs out of space; then it starts with the next row. Following are the various methods different from the BorderLayout:

getAlignment()	Gets the alignment for this layout.
setAlignment(int align)	Sets the alignment for this layout.

Consider the following example:

```
import java.awt.*;
import java.applet.Applet;
public class myButtons extends Applet {
Button button1, button2, button3;
public void init() {
button1 = new Button("Ok");
button2 = new Button("Open");
button3 = new Button("Close");
add(button1);
add(button2);
add(button3);
}
}
```

Fig 12.10 Flow Layout

The GridLayout class is used to lay out the components of a Container object in a grid where all components are the same size. The GridLayout constructor is used to specify the number of rows and columns of the grid. Following are the various methods different from the BorderLayout:

getColumns()	Gets the number of columns in this layout.
getRows()	Gets the number of rows in this layout.
setColumns(int cols)	Sets the number of columns in this layout to the specified value.
setRows(int rows)	Sets the number of rows in this layout to the specified value.

Consider the following example:

```
import java.awt.*;
import java.applet.Applet;
public class ButtonGrid extends Applet {
public void init() {
setLayout(new GridLayout(3,2));
add(new Button("1"));
add(new Button("2"));
add(new Button("3"));
add(new Button("4"));
add(new Button("5"));
add(new Button("6"));
}
}
```

The output of the above program is given below:

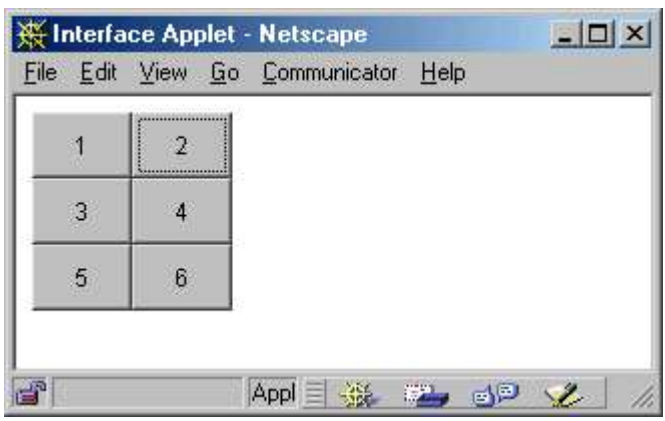


Fig 12.11 Grid Layout

This is the most powerful, complex, and hard-to-use Layout Manager that comes with the AWT. The GridBagLayout class lays out the components of a Container object in a grid-like fashion, where some components may occupy more than one row or column. The GridBagConstraints class is used to identify the positioning parameters of a component that is contained within an object that is laid out using GridBagLayout.

Using this layout you can place your object in CENTER, EAST, NORTH, NORTHEAST, NORTHWEST, SOUTH, SOUTHEAST, SOUTHWEST and WEST direction.

EVENT HANDLING

Java applications and applets run in an event-driven environment, which means that most actions that take place in Java generate an event that can be handled and responded to.

An event is something that happens when a program runs, and user events are things that a user causes by using the mouse, keyboard, or another input device. Responding to user events often is called event-handling.

Responding to user events in a Java program requires the use of one or more EventListener interfaces. As you might recall from using the Runnable interface for multithreaded programs, interfaces are special classes that enable a class of objects to inherit behavior that it would not be able to use otherwise.

Whenever, you have to add an EventListener interface to your application or applet, you must import the group of classes java.awt.event.* as all the listening classes are part of this group. To import these classes, you can add the following statement in your source code

import java.awt.event.*;

Also, you must use the implements statement with the class to declare that it will be using one or more listening interfaces. The following statement creates a class that uses ActionListener, an interface used with buttons and other components:

public class mylistner extends java.applet.Applet implements ActionListener {

The EventListener interfaces enable a GUI component to generate user events. For each component in the program, which should listen to the user inputs, you must add a listener interface. For example, to have the program respond to a mouse click on a button or the Enter key being pressed in a text field, you must include the ActionListener interface. To respond to the use of a choice list or check boxes, the ItemListener interface is needed. When you require more than one

interface, separate their names with commas after the implements statement. The following is an example:

```
public class mylist1 extends java.applet.Applet
implements ActionListener, MouseListener {
```

Following is the list of interfaces that can be used for various purpose

ActionListener	The listener interface for receiving action events.
AdjustmentListener	The listener interface for receiving adjustment events.
AWTEventListener	The listener interface for receiving notification of events dispatched to objects that are instances of Component or MenuComponent or subclasses.
ComponentListener	The listener interface for receiving component events.
ContainerListener	The listener interface for receiving container events.
FocusListener	The listener interface for receiving keyboard focus events on a component.
HierarchyBoundsListener	The listener interface for receiving ancestor moved and resized events.
HierarchyListener	The listener interface for receiving hierarchy changed events.
InputMethodListener	The listener interface for receiving input method events.
ItemListener	The listener interface for receiving item events.
KeyListener	The listener interface for receiving keyboard events (keystrokes).
MouseListener	The listener interface for receiving "interesting" mouse events (press, release, click, enter, and exit) on a component.
MouseMotionListener	The listener interface for receiving mouse motion events on a component.
TextListener	The listener interface for receiving text events.
WindowListener	The listener interface for receiving window events.

The Java AWT is responsible for generating events in response to user actions. This process of routing an event to a target object is known as posting an event. The method used to post events to target objects is called the postEvent method, which is defined for all target objects.

The postEvent method calls the handleEvent method for target objects derived from Component.

The handleEvent method serves as the default handler for all events, and it has the option of responding to an event or letting it pass through. If handleEvent doesn't handle an event, it returns false, in which case the parent object's handleEvent method is called. This process continues until an event is handled or the top of the object tree is reached.

All types of Events that can occur in the system are defined in Event Class. The Event class is used primarily by the handleEvent method, which is defined as:

```
public boolean handleEvent(Event evt)
```

Once handleEvent method gets the information about the event occured, it calls a more specific event-handler method to deal with the event. For example, if a mouse is pressed, the Event Object's id memory variable is set to MOUSE_DOWN, which is a constant defining the Mouse down event.

Now, handleEvent checks the value of id and upon finding it equal to MOUSE_DOWN, calls the mouseDown handler method.

```
public boolean handleEvent(Event evt) {
switch (evt.id) {
    ...
    case Event.MOUSE_DOWN:
    return keyDown(evt, evt.key);
    ...
}
return false;
}
```

The Event class is a critical component of the Java AWT. Event objects are constructed and passed into methods such as postEvent when an event occurs.

The Event class defines a set of constants, which specify the different types of possible events. The partial list of constants is given below, which is quite self-explanatiory:

```
static int KEY_ACTION
static int KEY_ACTION_RELEASE
static int KEY_PRESS
static int KEY_RELEASE
static int LEFT
static int LIST_DESELECT
static int LIST_SELECT
```

static int LOAD_FILE
static int LOST_FOCUS
static int META_MASK
static int MOUSE_DOWN
static int MOUSE_DRAG
static int MOUSE_ENTER
static int MOUSE_EXIT
static int MOUSE_MOVE
static int MOUSE_UP
static int NUM_LOCK
static int PAUSE
static int PGDN
static int PGUP
static int PRINT_SCREEN
static int RIGHT
static int SAVE_FILE
static int SCROLL_ABSOLUTE
static int SCROLL_BEGIN
static int SCROLL_END
static int SCROLL_LINE_DOWN
static int SCROLL_LINE_UP
static int SCROLL_LOCK
static int SCROLL_PAGE_DOWN
static int SCROLL_PAGE_UP
static int SHIFT_MASK
static int TAB
Object target
static int UP
long when
static int WINDOW_DEICONIFY
static int WINDOW_DESTROY
static int WINDOW_EXPOSE
static int WINDOW_ICONIFY
static int WINDOW_MOVED
int x
int y

The id field is used by the AWT to distinguish between event types. The id field indicates what type of event it is and which other Event variables are relevant for the event.

The Event class offers the following methods:

boolean controlDown()	Checks if the Control key is down.
boolean metaDown()	Checks if the Meta key is down.

protected String paramString()	Returns the parameter string representing this event.
boolean shiftDown()	Checks if the Shift key is down.
String toString()	Returns a representation of this event's values as a string.
void translate(int x, int y)	Translates this event so that its x and y coordinates are increased by dx and dy, respectively.

EXERCISE

1. Why are action events called by that name?
   (a) They occur in reaction to something else.
   (b) They indicate that some kind of action should be taken in response.
   (c) They honor cinematic adventurer Action Jackson.
2. What does this signify as the argument to an addActionListener() method?
   (a) "This" listener should be used when an event occurs.
   (b) "This" event takes precedence over others.
   (c) "This" class of objects will handle the events.
3. Fill in the Blanks:
   a) Text on a button can be known using the method and it can be changed using the method.
   b) _____ method prints the component and all of its subcomponents
   c) The layout manager arranges the components in a
   d) _____ method returns the preferred size of the component.
   e) _____ class is used to define a group for the radio buttons.
   f) _____ method returns the foreground color of the component.
   g) The Canvas class implements a _____ that supports drawing.
4. Describe layouts provided by any two layout managers.
5. Explain any two container objects.
6. Distinguish between the list and the choice control.
7. State True or False:
   1. Component class implements graphical user interface controls.
   2. When a choice selection is made, a MOUSE_EVENT is generated
   3. List control allows you to select multiple items at one time.
   4. The size of the frame includes any area designated for the border.
   5. Panel is placed over a layout.
8. Do you need to do anything with the paint() or repaint() method to indicate that a text field has been changed?

APPENDIX

DIFFERENCE BETWEEN JAVA ANC C++
IMPORTANT WEBSITE OF JAVA

Difference between Java anc C++

If you have the knowledge of C++, you will realize that most of the Java Syntax are similar to the C++ syntax. But still many a features of C++ were left out in Java and many new features were added to Java to take it beyond C++. Basically the differences exist in Program Structure, Program development approxach and Language Syntax.

Both Java and C++ uses main() function as the entry point for the program. But the difference in the two function is that while C++ function takes two parameters : int argc - to identify the number of arguments being passed and argv[][] - the character array that contains the program arguments, the Java main() function only takes takes a single args[] parameter of the String class. The number of arguments passed via the main() method invocation is determined by args.length.

Another major difference in two languages is that while Java supports the package approach, C++ doesn't support it. Java strictly adheres to the class-oriented approach.

Following are the major differences in the Program development area and Language Syntax Area:

a) C++ programs are generally compiled into native machine object code, whereas Java programs are compiled into the bytecode instructions of the Java virtual machine

b) Complied code of C++ is executed directly as a process running under the local operating system whereas compiled code of Java is executed using the Java interpreter or a Java-compatible browser.

c) Java has a rich and ever-expanding set of classes, which is extensively used for program development. The C and C++ codes can be used by the Java using native methods but Java rich set of classes, can't be used with standard C and C++ libraries.

d) C++ source code files are processed by a preprocessor before they are submitted to the compiler, but Java doesn't uses the preprocessor.

e) Java and C++ uses same symbol for comments, but Java has additional facility for insert doc commands which can be used by the javadoc program to create documentation.

f) Java and C++ classes are declared using a similar, but different, syntax - In C++ nesting of classes is allowed, but not in Java. Java classes support single inheritance, whereas C++ classes supports multiple inheritance. C++ supports templates. Java does not.

g) A Java variable contains the value of a primitive data type or refers to a Java object whereas the C++ variables doesn't have any restriction like that.

h) C++ supports pointers to the other objects, whereas Java doesn't support pointers.

i) Java objects are instances of classes or arrays. C++ objects do not include arrays.

j) In case of Arrays, both have similar arrays but there is a major difference. C++ supports true multidimensional arrays, where as supports only single dimensional arrays. The multidimensional array in Java is defined as arrays of arrays.

j) In control statements, Java and C++ differ in Goto statement, Java doesn't support goto statement. Java also provides the synchronized statement to support multithreading operations on critical sections. C++ does not support a synchronized statement.

k) In exception handling, C++ doesn't support the finalize clause whereas, Java does support.

Core Java

Everything you wanted to know about.....

This book provides a solid theoretical foundation with practical approach for Java Language. Authors discusses the key concepts of Java Language and present it in bite-sized information for quick and easy reference. The concepts has been explained in plain simple English thus ensuring your success. It also covers real-world analogies to help you understand the various aspect of Language.

HIGHLIGHTS

- Explains various Programming Concepts
- Learn to use Java for developing applets and applications
- Learn by doing, Ideal for Learners
- Gives step by step guidance developing codes
- Packed with numerous real world source codes
- Numereous screen illustrations to help you at every step

About the Authors

Munishwar Gulati is a graduate from world famous University of Roorkee and has about 10 years of experience in computer industry. He has written many computer based titles, from DOS , to COBOL, to desktop publishing, to Visual Studio, to Web designing and many more, for various training centres all over India.
Co-Author Mini Gulati is also a graduate in Computer Science and has 6 years rich experience in developing softwares on various platforms. She is also co-authoring many of the up-coming titles.

ISBN 81-87870-13-3

SILICON MEDIA PRESS™

www.siliconmedia.org