Chapter 1 - Object Oriented Concepts

With the increasing complexity of software ,it is just not enough to put together a sequence of programming statements and sets of procedures ,modules. We need to use sound construction techniques such as modular programming, top down programming, bottom up programming and structured programming.

With the advent of languages such as C, structured programming became very popular in 1980s. It proved to be a powerful tool that enabled programmers to write moderately complex programs fairly easy. However as the programs grew larger, the structured approach failed to show the desired results in terms of bug free, easy to maintain and reusable programs.

Object oriented programming (OOP) is an approach to program organization and development, which attempts to eliminate some of the pitfalls of conventional programming methods by incorporating the best of structured programming features with several new concepts. It is a new way of organizing and developing programs and has nothing to do with any particular language. Languages that

support OOP features include Smalltalk, Objective C, C++, Ada and Object Pascal. C++, an extension of C language, is the most popular OOP language today. C++ is basically a procedural language with object oriented extension. The latest one added to this list is Java, a pure object-oriented language.

1.1 BASIC CONCEPTS OF OBJECT-ORIENTED PROGRAMMING

The major object-oriented approach is to eliminate some of the flaws encountered in the procedural approach. OOP treats data as critical element in the program development and does not allow it to flow freely around the system.

Oops allows us to decompose a problem into a number of entities called objects and then build data and functions (known as methods in Java) around these entities.

It ties data more closely to the functions that operate on it & protects it from accidental modifications from outside functions.

Objects may communicate with each other through functions. i.e. functions of one object can access the functions of other object but data of an object can be accessed only by the functions associated with the object.

An Object is considered to be a partitioned area of computer memory that stores data & set of operations that can access that data. Since memory partitions are independent the objects can be used in a variety of different programs without modifications.

1.1.1 Objects and Classes

Objects are basic runtime entities, in an object oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program may handle. They may also represent user defined data types like lists or arrays.

The objects contain data and code to manipulate that data. The entire set of data and code of any object can be made a user-defined data type using the concept of a class. A class may be thought of as a ‘data type’ and an object as a ‘variable’ of that data type. Once the class has been defined, we can create any number of

objects belonging to that class. Each object is associated with the data of type class with which they are created. e.g. Mango, apple and orange are the members of thee class fruit.

1.1.2 Data abstraction and encapsulation

The wrapping up of data and methods into a single unit (Called class) is known as encapsulation. Data encapsulation is the most striking feature of class. The data is not accessible to the outside world and only those methods , which are wrapped in the class, can access it. These methods provide the interface between the object's data and the program. The encapsulation of the data from direct access by the program is called data hiding. Encapsulation makes it possible for objects to be treated like 'black boxes', each performing a specific task without any concern for internal implementation.

Abstraction refers to the act of representing essential features without including the background details or explanations.

1.1.3 Inheritance

Inheritance is the process by which objects of one class acquire the properties of objects of another class. Inheritance supports the feature of hierarchical classification. The concepts of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from an existing one. In Java the main class is known as 'super class' and derived class is known as 'sub class'.

1.1.4 Polymorphism

Polymorphism means ability to take more than one form. E.g. an operation may exhibit different behavior in different instances. The behavior depends on type of data used in the operation. E.g. Consider the operation of addition. For two numbers , it will generate sum. For two strings it will generate third string which is concatenation of current two strings. A single function name can be used to handle different number and different types of arguments.

Polymorphism plays an important role in allowing objects having different internal structures to share thee same external interface. This means that a general class of operation may be accessed in the same manner even though specific actions associated with each operation may differ. Polymorphism plays an important role. Polymorphism is extensively used in implementing inheritance.

1.1.5 Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at runtime. It is associated with polymorphism and inheritance.

1.1.6 Message Communication

An object oriented program consists of a set of objects that communicate with each other. Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. A message for an object is a request for execution of a procedure and will invoke a method (procedure) in the receiving object that generated the desired result.

Message passing involves specifying the name of the object, the name of the method(message) and the information to be sent. E.g.

```
Employee.salary(name);
```

Here employee is the object, salary is the message and name is the parameter that contains information.

Chapter 2 - JAVA FEATURES

Java is a general purpose, object-oriented programming language developed by sun microssystems of usa in 1991.

2.1 JAVA FEATURES

The inventors of Java wanted to design a language which could offer solutions to some of the problems encountered in modern programming.

2.1.1 Compiled And Interpreted

USUALLY A COMPUTER LANGUAGE EITHER COMPILED OR INTERPRETED. Java combines both these approaches thus making Java a two-stage system. First, Java compiler translates source code into what is known as bytecode instructions. Bytecodes are not machine instructions and therefore, in the second stage, Java interpreter generates machine code that can be directly executed by the machine that is running the Java program.

2.1.2 Platform-Independent and Portable

The most significant contribution of Java over other languages is its portability. Java programs can be easily moved from one computer system to another, anywhere and anytime. Changes and upgrades in operating systems, processors and system resources will not force any changes in Java programs. This is the reason why Java has become a popular language for programming for internet which interconnects different kinds of systems worldwide. We can download a Java applet from a remote computer onto our local system via internet and execute it locally. This makes the internet an extension of the user's basic system providing practically unlimited number of accessible applets and applications.

Java ensures portability in two ways. First, Java compiler generates bytecode instructions that can be implemented on any machine. Secondly, the size of the primitive data types are machine independent.

2.1.3 Object Oriented

Java is a true object-oriented language. Almost everything in Java is an object. All program code and data reside within objects and classes. Java comes with an extensive set of classes, arranged in packages, that we can use in our programs by inheritance. The object model in Java is simple and easy to extend.

2.1.4 Robust and Secure

Java is a robust language. It provides many safeguards to ensure reliable code. It has strict compile time and run time checking for data types. It is designed as a garbage collected language relieving the programmers virtually all memory management problems.

Security becomes an important issue for a language that is used for programming on internet.

2.1.5 Distributed

Java is designed as a distributed language for creating applications on networks. It has the ability to share both data and programs. Java applications can open and access remote objects on internet as easily as they can do in a local system.

2.1.6 Familiar, Simple and Small

Java does not use pointers, preprocessor header files, goto statement and many others. It also eliminates operator overloading and multiple inheritance. It is modeled on c and c++ languages, so looks familiar.

2.1.7 Multithreaded

Multithreaded means handling multiple tasks simultaneously. Java supports multithreaded programs. This means that we need not wait for the application to finish one task before beginning another. This feature greatly improves the interactive performance of graphical applications. The Java runtime comes with tools that support multiprocess synchronization and construct smoothly running interactive systems.

2.1.8 High Performance

Java performance is impressive for an interpreted language, mainly due to the use of intermediate bytecode. Java architecture is also designed to reduce overheads during runtime. Further, the incorporation of multithreading enhances the overall execution speed of Java programs.

2.1.9 Dynamic and Extensible

Java is a dynamic language. Java is capable of dynamically linking in new class libraries, methods and objects. Java can also determine the type of class through a query, making it possible to either dynamically link or abort the program, depending on the response.

Java programs support functions weitten in other languages such as c and c++. These functions are known as native methods. This facility enables the programmers to use the efficient functions available in these languages. Native methods are linked dynamically at runtime.

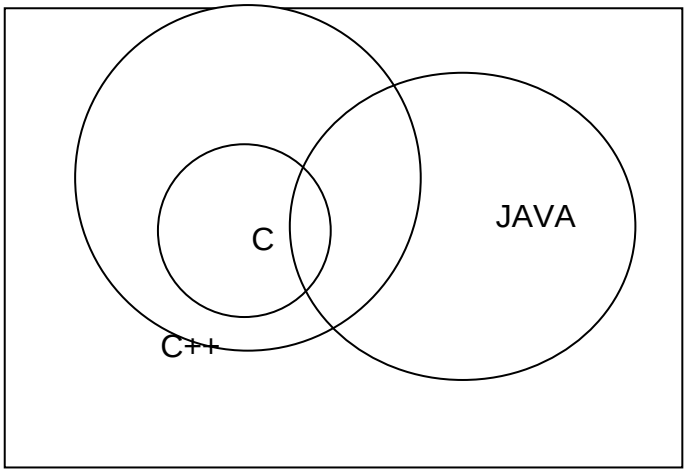
2.1.10 Java and C Differences

- Java does not include the C unique statement keywords goto, sizeof and typedef.
- Java does not contain the data types struct, union and enum.
- Java does not define the type modifiers keywords auto, extern, register, signed and unsigned.
- Java does not support an explicit pointer type.
- Java does not have a preprocessor and therefore we cannot use #define, #include and #ifdef
- Java does not support any mechanism for defining variable arguments to functions.
- Java requires that the functions with no arguments must be declared with empty parenthesis and not with the void keyword as done is C.
- Java adds new operators such as instanceof and >>>.

- Java adds labeled break and continue statements.
- Java adds many features required for object oriented programming.

2.1.11 Java and C++ Differences

- Java does not support operator overloading
- Java does not have template classes as in C++
- Java does not support multiple inheritance in classes. This is accomplished using a new feature called “Interface”
- Java does not support global variables. Every variable and method is declared within a class and forms part of that class.
- Java does not use pointer.
- Java has replaced the destructor function with a finalize() method.
- There are no header files in Java



Overlapping of C, C++ and JAVA

2.2 JAVA ENVIRONMENT

Java environment includes a large number of development tools and hundred of classes and methods. The development tools are part of the system knows and Java Development Kit (JDK) and the classes and methods are part of the Java Standard Library (JSL), also known as the Application Programming Interface (API).

2.2.1 Java Development Kit

The Java Development Kit comes with a collection of tools that are used for developing and running Java programs.

Tool	Description
Appletviewer	Enables us to run Java applets (without using Java compatible browser)
Java	Java interpreter which runs applets and applications by reading and interpreting bytecode files.
Javac	The Java compiler, which translates Java source code to bytecode files that the Interpreter can understand.
Javadoc	Creates html format documentation from Java source code files
Javah	Produces header files for use with native methods
Javap	Java disassembler, which enables us to convert bytecode files into a program description
Jdb	Java debugger, which helps us to find errors in our programs.

The following figure shows the way these tools are applied to build and run application programs.

THE PROCESS OF BUILDING AND RUNNING JAVA APPLICATION

2.3 APPLICATION PROGRAMMING INTERFACE

The Java standard library (or API) includes hundreds of classes and methods grouped into several functional packages. These packages together contain more that 1500 classes and interfaces and define more than 13,000 methods.

Most commonly used packages are :

- Language Support package – A collection of classes and methods required for implementing basic features of Java.
- Utilities Package – A collection of classes to provide utility functions such as date and time functions.

Input/ output package – A collection of classes required for input/output manipulation.

- Networking Package – A collection of classes for communicating with other computer via internet.
- AWT package – The abstract window tool kit package contains classes that implements platform independent graphical user interface.
- Applet Packge – this includes a set of classes that allows us to create Java applets.

Chapter 3 - overview of java language

3.1 INTRODUCTION

Java is a general purpose, object oriented language. We can develop two types of Java programs:

- Standalone applications
- Web applets

Standalone Applications are programs written in Java to carry out certain tasks on a standalone local computer. In fact Java can be used to develop programs for all kinds of applications. Executing a standalone Java program involves two steps.

1. Compiling source code into Bytecode using javac compiler
2. Executing the Bytecode program using Java interpreter

Applets are small Java programs developed for Internet applications. An applet located on a distant computer (server) can be downloaded via Internet and executed on a local computer (client) using a Java-capable browser. We can develop applets for doing everything from simple animated graphics to complex games and utilities. Since applets are embedded in an HTML (Hypertext Markup Language) document and run inside a web page, creating and running applets are more complex than creating application.

Stand-alone Java programs can read and write files and perform certain operations that applets can not do. An applet can only run within a web browser.

3.2 SIMPLE JAVA PROGRAM

```
class Sampleone
{
        public static void main(String args[])
        {
                System.out.println(“Hello World !”);
        }
}
```

The best way to learn is a hello world example.

This is the simplest Java program. Nevertheless, it brings out the basic features of the language. So we will discuss the program line by line and understand the unique features that constitute a Java program.

Since Java is a case sensitive language, we need to type in the same case.

3.2.1 Class Declaration

The first line

```
class SampleOne
```

Declares a class, which is an object-oriented construct. As stated earlier, Java is true object-oriented language and therefore, everything must be placed inside a class. Class is a keyword and declares that a new class definition follows. SampleOne is a Java Identifier that specifies the name of the class to be defined.

3.2.2 Opening Brace

Every class defination in Java begins with an opening brace "{" and ends with a matching closing brace "}", appearing in the last line in that example.

3.2.3 The main line

The third line

```
Public static void main(String args[])
```

Defines a method named main. This is similar to the main() function in C,C++.

Every Java application program must include the main() method. This is starting point for the interpreter to begin the execution of the program. A Java application can have any number of classes but only one of them must include a main method to initiate the execution. (Java applets will not use the main method at all)

This line also includes following keywords.

Public	The keyword public is an access specifier that declares the main method as unprotected and therefore making it accessible to all other classes.

Static	Next appears the keyword static, which declares this method as one that belongs to the entire class and not a part of any objects of the class. The main must always be declared as static since the interpreter uses this method before any objects are created.
Void	The type modifier void states that them main method does not return any value (but simply prints some text to the screen.)

All parameters to a method are declared inside a pair of parentheses. Here, String args[] declares a parameter names args, which contains an array of objects of the class type String.

3.2.4 The output Line

The only executable statement in the program is

System.out.println("Hello World !");

Since Java is true object oriented language, every method must be part of an object. The println method is a member of the out object, which is a static data member of System class. The line prints the string

Hello World !

to the screen. The method println always appends a new line character to the end of the string. **Every Java statement must end with a semicolon.**

3.3 MORE OF JAVA

Assume that we would like to compute and print the square root of a number. A Java program to accomplish this is shown on next page.

3.3.1 Program Details

The statement

Double x = 5 ;

Declares a variable x ans intializes it to the value 5 and the statement

Double y ;

Merely declares a variable y. The statement

```
/*
* More java statements
* A program to compute square root
*/
import java.lang.Math;
class SquareRoot
{
        public static void main(String args[])
        {
                double x = 5;
                double y ;
                y = Math.sqrt(x);
                System.out.println("Y = " +y);
        }
}
```

Y = math.sqrt(x);

Invoke the method sqrt of the Math class, computes square root of x and then assigns the result to the variable y. The output statement

System.out.println(“ y = “+y);

Displays the result on the screen as

Y = 2.236067...

Note the use of + symbol. Here it acts as the concatenation operator of two strings. The value of y is converted into a string representation before concatenation.

3.3.2 Use of Math Function

Note that the first statement in the program is import java.lang.Math;

The purpose of this statement is to instruct the interpreter to load the math class from the package lang. Math class contains the sqrt method required in the program.

3.3.3 Comments

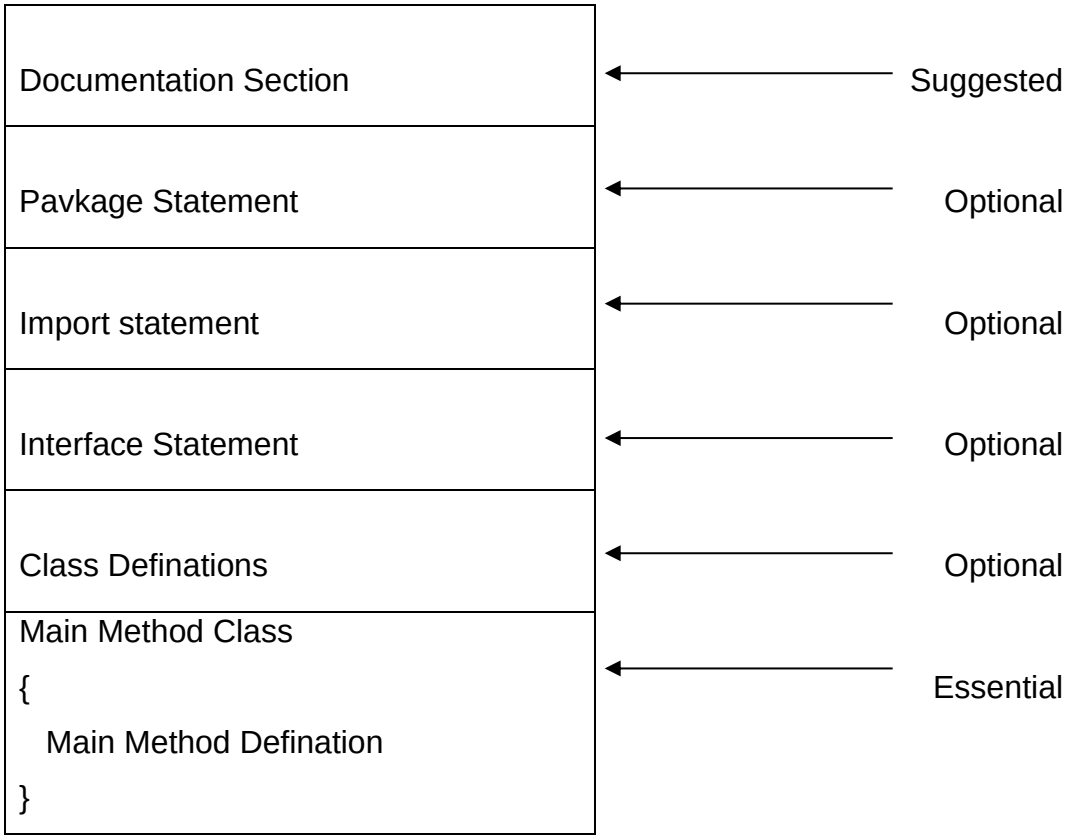
Java permits both the single-line comments and multi-line comments. Single line comments begin with // and end at the end of the line as shown on the lines declaring x and y. for longer comments, we can create long multi-line comments by starting with a /* and ending with */.

3.4 AN APPLICATION WITH TWO CLASSES

Both the examples discussed above use only one class that contains the main method. A real-life application will generally require multiple classes.

```
class Room
{
        float length;
        float breadth;

        void getdata(float a, float b)
        {
                length  = a;
                breadth= b;
        }
}

class RoomArea
{
        public static void main(String args[])
        {
                float area;
                Room room1 = new Room(); // Creates an object room1
                room1.getdata(14,10);
                area = room1.length * room1.breadth;
                System.out.println("Area = "+area);
        }
}
```

3.5 JAVA PROGRAM STRUCTURE



3.5.1 Documentation section

The documentation section comprises a set of comment lines giving the name of the program, the author and other details, which the programmer would like to refer to at a later stage. Comments must explain why and what of classes and how of algorithms. In addition to the two styles of comments discussed earlier, Java also uses a third style of comment /** …*/ known as documentation comment. This form of comment is used for generating documentation automatically.

3.5.2 Package Statement

The first statement allowed in a Java file is a package statement. This statement declares a package name and informs the compiler that the classes defined here belong to this package.

```
Package student;
```

The package statement is optional. That is, our classes do not have to be part of a package.

3.5.3 Import Statements

The next thing after a package statement may be number of import statements. This statement instructs the interpreter to load the class or classes from given package.

3.5.4 Interface Statements

An interface is like a class but includes a group of method declarations. This section is used only if there is a need to implement the multiple inheritance feature.

3.5.5 Class Definations

A Java program may contain multiple class definations. Classes are the primary and essential elements of a Java program. These classes are used to map the objects of real-world problems. The number of classes used depends on the complexity of the problem.

3.5.6 Main method class

Every Java standalone program requires a main method as its starting point. A simple Java program may contain only this part. The main method creates objects of various classes and establishes communications between them. On reaching the end of main, the program terminates and the control passes back to the operating system.

3.5.7 Java tokens

A Java program is basically collection of classes. A class is defined by a set of declaration statements and methods containing executable statements. Most statements contain expressions, which describe the actions carried out on data. Smallest individual units in a program are known as tokens. The compiler recognizes them for building up expressions and statements.

A java program is collections of tokens, comments and white spaces. Java language includes five types of tokens. They are

- Reserved Keywords
- Indentifiers
- Literals
- Operators
- Separators

3.5.8 Java character set

The smallest units of Java language are the characters used to write Java tokens. These characters are defined by the Unicode character set, an emerging standard that tries to create characters for large number of scripts worldwide. The Unicode is a 16-bint character coding system and currently supports more than 34,000 defined characters derived from 24 languages.

3.5.9 Keywords

Keywords are an essential part of a language defination. They implement specific features of the languages. Java has reserved 60 words as keywords. Since keywords have specific meaning, we can not use them as names for variables.
e.g.

abstract, boolean, byte, case, catch, char, class, static, super, switch etc

3.5.10 Identifiers

Identifiers are programmer-designed tokens.

They are used for naming classes, methods, variables, objects, labels, packages and interfaces in a program. Java identifiers follow following rules.

They can have alphabets, digits and the underscore and dollar sign characters.

They must not begin with a digit.

Uppercase and lowercase letters are distinct

The can be of any length

Identifiers must be meaningful, short enough to be quickly and easily types and long enough to be descriptive and easily read. Java developers have followed some naming conventions

Names of all public methods and instance variables start with a leading lowercase letter. E.g. average, sum

When more than one word is used in a name, the second words are marked with leading uppercase letter. E.g. dayTemperature, totalMark

All private and local variables use only lowercase letter combined with underscore. E.g. length, batch_strength

All classes and interfaces start with a leading uppercase letter e.g. Student, HelloJava

Variable that represent constant values use all upperscore letters and underscores between words. E.g. TOTAL, F_MAX

All these are conventions and not rules.

3.5.11 Literals

Literals are a sequence of characters (digits, letters and other characters) that represent constant values to be stored in variables. Java language specifies five major types of literals.

Integer Literals, Floating_point literals, Character literals, String literals, Boolean literals

3.5.12 Operators

An operator is a symbol that takes one or more arguments and operates on them to produce a result.

3.5.13 Separators

Separators are symbols used to indicate where the group of code are divided and arranged. They basically define the shape and function of our code. Eg. Parentheses (), braces {}, brackets [], semicolon; etc

3.5.14 Java statements

The statements in Java are like sentences in natural language. A statement is an execution combination of tokens ending with a semicolon (;) mark. Statements are usually executed in sequence in the order in which they appear.
The next page contains the classification of Java statements. The implementation of these statements is same like C and C++.

3.6 IMPLEMENTING A JAVA PROGRAM

Implementation of a Java application program involves a series of steps. They include

- Creating the program
- Compiling the program
- Running the program

3.6.1 Creating the Program

The program must be saved with Java extension. This file is called as source file. All Java source files will have extension as “java”. Note also that if a program contains multiple classes, the file name must be the classname of the class containing the main method or the public class.

3.6.2 Compiling the Program

To compile the program, we must run the Java compiler javac, with the name of the source file on the command line as shown below.

javac Test.java

If every thing is ok, the javac compiler creates a file called Test.class containing the bytecode of the program. Note that the compiler automatically names the bytecode file as <classname>.class.

3.6.3 Running the Program

We need to use the Java interpreter to run a stand-alone program. At the command prompt, type

java Test

Now, the interpreter looks for the main in the program and begins execution from there. When executed, the program displays results.

3.6.4 Machine neutral

The compiler converts the source code files into bytecode files, these codes are machine independent and therefore can be run on any machine. That is a program compiled on IBM machine will run on a Macintosh machine.

Java interpreter reads the bytecode files and translates them into machine code for the specific machine on which the Java program is running. The interpreter is written for each type of machine.

3.7 JAVA VIRTUAL MACHINE

All languages compilers translate source code into machine code for a specific computer. Java compiler also does the same thing. Then how does Java achieve architecture neutrality? The answer is that the Java compiler produces an intermediate code known as bytecode for a machine that does not exist. This machine is called as the Java virtual machine and it exists only inside the computer memory. It is a simulated computer within the computer and does all major functions of a real computer.

The virtual machine code (byte code) is not machine specific. The Java interpreter generates the machine specific code (known as machine code) by acting as intermediary between the virtual machine and the real machine.

3.8 COMMAND LINE ARGUMENTS

There may be occasions when we may like our program to act in a particular way depending on the input provided at the time of execution. This is achieved in Java programs by using what are known as command line arguments. Command line arguments are parameters that are supplied to the application program at the time of invoking it for execution.

Consider the command line

JAVA ComLineTest BASIC FORTRAN C++ JAVA

The command line contains four arguments. These are assigned to the array as follows

BASIC	-	args[0]
FORTRAN	-	args[1]
C++	-	args[2]
JAVA	-	args[3]

Below is a program using command line arguments.

```
/* This program uses command line argumnts */
class ComLineTest
{
 public static void main(String args[ ])
{
    int count,i=0;
    String string;
    count=args.length;
    System.out.println("Number of arguments = " +
count);
    while(i<count)
   {
       string=args[i];
       i=i+1;
       System.out.println(i + " : " + " Java is "
+string+"!");
    }
}
}
```

Chapter 4 - Constants,Variables and Data Types

Like any other language, Java has its own vocabulary and grammar. In this chapter, we will discuss the concepts of constants and variables and their types as they relate to Java language.

4.1 CONSTANTS

Constants in Java refer to fixed values that do not change during the execution of a program. Java supports several types of constants.

4.1.1 Integer Constants

123 -321 0 678932

4.1.2 Real Constants

0.00034 -0.55 435.222

4.1.3 Single Character Constants

’5’ ‘x’ ‘;’ ‘ ‘

4.1.4 String Constants

“Hello Java” “2000” “5+4”

4.1.5 Backslash Character Constants

Java supports some special backslash character constants that are used in output methods. A list of such backslash character constants is as follows.

Constants	Meaning
‘\b’	back space
‘\f’	form feed
‘\n’	New line
‘\r’	carriage return
‘\t’	horizontal tab
‘\’’	single quote
‘\”’	double quote
‘\\’	backslash

4.2 VARIABLES

A variable is an identifier that denotes a storage location used to store a data value. Unlike constants that remain unchanged during the execution of a program, a variable may take different values at different times during the execution of the program.

The programmer can choose a variable name in a meaningful way so as to reflect what it represents in the program. Eg. Average, height, total_height etc

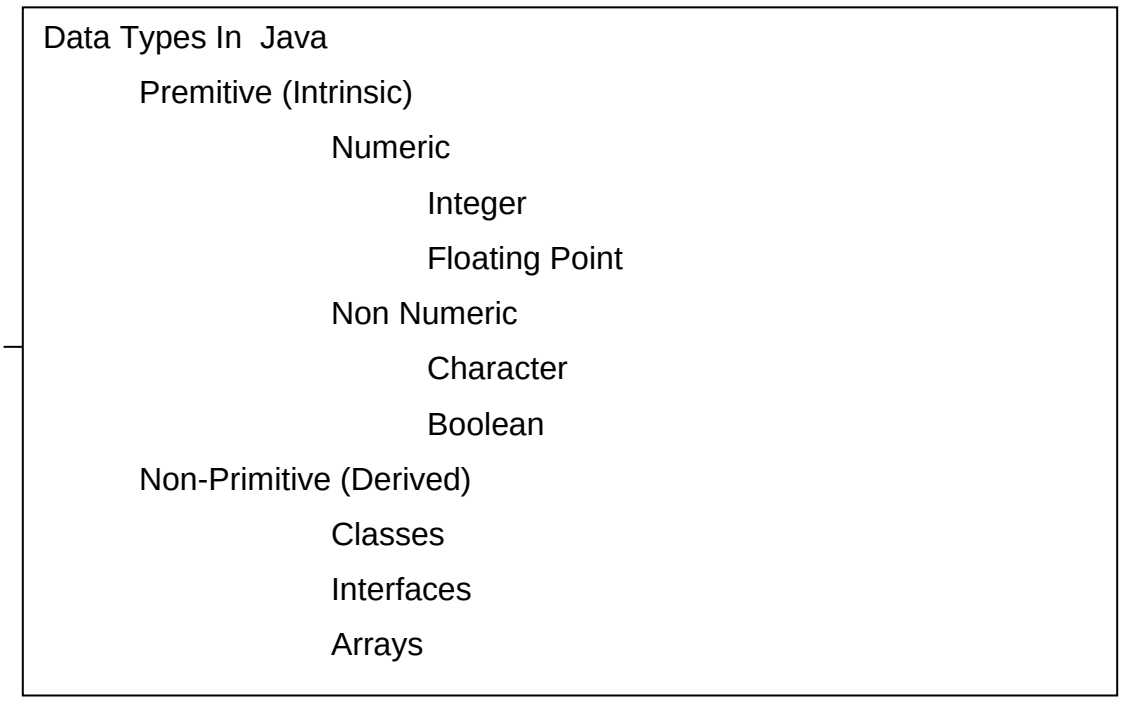
Variable names may consist of alphabets, digits, the underscore (_) and dollar characters with following conditions:

- They must not begin with a digit.

- Uppercase and lowercase are distinct. This means Total is not same as total or TOTAL.
- It should not be a keyword.
- White space is not allowed.
- Variables names can be of any length.

4.3 DATA TYPES

Every variable in Java has a data type. Data types specify the size and type of values that can be stored. Java language is rich in its data types. Following table shows main data types with categories.



4.3.1 Integer Types

Integer types can hold whole numbers such as 123, -98 and 4567. They further divided into subtypes as follows:

Type	Size	Range
Byte	one byte	-128 to 127
Short	two bytes	-32,768 to 32,767
Int	four bytes	-2,147,483,648 to 2,147,483,647
Long	Eight bytes	-9,223,372,036,854,808 to 9,223,372,036,854,807

4.3.2 Floating Point Types

Floating point type can hold number with decimal points such as 21.34 –1.234.

Type	Size	Range
Float	four bytes	3.4e-038 to 3.4e+038
Double	eight bytes	1.7e+308

4.3.3 Character Type

In order to store character constants in memory, Java provides a character data type called char. The char type size assumes size 2 bytes, but it holds only single character.

4.3.4 Boolean Type

Boolean type is used when we want to test a particular condition during the execution of the program. There are only two values that a boolean type can take – true and false. Both these are keywords. It uses only one bit of storage.

4.4 DECLARATION OF VARIABLES

Variables are the names of storage locations. After designing suitable variable names, we must declare them to the compiler. Declaration does three things

- It tells the compiler what the variable name is.
- It specifies what type of data the variable will hold.
- The place of declaration decides the scope of the variable.

A variable must be declared before it is used in the program.

A variable can be used to store a value of any data type. The general form of declaration of a variable is

type variable, variable, ….,variableN;

e.g.

```
int count;
float x, y;
```

4.5 GIVING VALUES TO VARAIBLES

A variable must be given a value after it has been declared but before it is used in an expression. This can be achieved in two ways

- By using an assignment statement
- By using a read statement

4.5.1 Assignment statement

A simple method of giving value to a variable is though the assignment statement as follows

```
variablename = value;
```

e.g.

```
initialvalue = 0;
yes = ‘x’;
```

We can also string assignment expressions as shown below

X = y = z =0;

It is also possible to assign a value to a variable at the time of its declaration. It takes the form

```
Type variablename = value;
```

e.g. int finalvalue = 100;

char yes = ‘x’;

The process of giving initial values to variables is known as initialization. The ones that are not initialized are automatically set to zero.

4.5.2 Read Statement

We can give values to variables through keyboard using the readLine() method as follows

The readLine() method (which is invoked using an object of class DataInputStream) reads the input from the keyboard as a string which is then converted in to corresponding data type using the data wrapper classes.

Note that we have used the keywords try and catch to handle any errors that might occur during the reading process.

```
import  java.io.DataInputStream;
class Reading
{
   public static void main(String args[ ])
          {
          DataInputStream in = new DataInputStream(System.in);
          int intNumber=0;
          float floatNumber = 0.0f;
          try
          {
                    System.out.println("Enter an Integer : ");
                    intNumber=Integer.parseInt(in.readLine());
                    System.out.println("Enter a float Number: ");
                    floatNumber = Float.valueOf(in.readLine()).floatValue();
          }
          catch(Exception e) {}

          System.out.println("intNumber = "  +  intNumber);
          System.out.println("floatNumber = "  +  floatNumber);
          }
}
```

4.6 SCOPE OF VARIABLES

Java variables are actually classified into three kinds

- Instance variables

- Class variables
- Local variables

Instance and Class variables are declared inside a class. Instance variables are created when the objects are instantiated and therefore they are associated with the objects. They take different values for each object. On the other hand, class variables are global to a class and belong to the entire set of objects that class creates. Only one memory location is created for each class variable.

Variables declared and used inside methods are called local variable. They are called so because they are not available for use outside the method defination. Local variables can also be declared inside program blocks that are defined between an opening brace { and a closing brace }. These variables are visible to the program only from the beginning of its program block to the end of the program block. When the program control leaves a block , all the variables in the block will cease to exist. The area of the program where the variable is accessible(usable) is called its scope. We can have program blocks within other program blocks which is called nesting. We cannot declare a variable to have the same name as one in an outer block. (which was valid in C, C++)

4.7 SYMBOLIC CONSTANTS

We often use certain unique constants in a program. These constants may appear repeatedly in a number of places in the program. E.g. pi or max number etc. We face two problems in the subsequent use of such programs. They are

- Problem in modification of the program.
- Problem in understanding the program

A constant is defined as follows:

```
final datatype symbolic-name = value;
```

Example:

```
final  float  PI=3.14;
final   int    SIZE=20;
```

In above declaration final is keyword.

Note that:

- Symbolic names take the same form as variables names. But they are written in CAPITALS to visually distinguish them from normal variable names.
- They can not be declared inside a method. They should be used only as class data members in the beginning of the class.

4.8 TYPE CASTING

We often encounter situations where there is a need to store a value of one type into a variable of another type. In such situations, we must cast value to be stored by proceeding it with parentheses.

The syntax is:

```
type variable1= (type) variable2;
```

The process of converting one data type to another is called casting.

Example:

```
int m = 50;
byte n=(byte) m;
long count = (long)m;
```

Casting into smaller type can result in a loss of data. Casting of float into int will result in loss of fractional part. The list of casts, which are guaranteed to no loss of data as follows.

From	To
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

4.8.1 Automatic Convertion

For some types, it is possible to assign a value of one type to a variable of different type without a cast. Java does the conversion of assigned value automatically. This is known as automatic type conversion. This is possible only if destination type has enough precision to store the source value. E.g. int is large enough to hold a byte value. Therefore,

```
Byte b = 75;
int a = b;
```

are valid statements.

The process of assigning a smaller type to a larger type is called widening or promotion and reverse is called narrowing.

4.9 PRINTING VALUES OF VARIABLES

A computer program is written to manipulate a given set of data and to display or print the results. Java supports 2 methods that can be used to send results to the screen.

```
print( ) method          //print and wait
println( ) method        //print a line and move to next line.
```

The print() method sends information into buffer. This buffer is not flushed until a newline character is sent.

As a result, print() method prints output on one line until a newline character is encountered.

```
class Displaying
{
  public static void main(String args[])
{
    System.out.println("Screen Display");
    for(int i=0;i<=9;i++)
    {
      for(int j=0;j<=i;j++)
       {
         System.out.print("  ");
         System.out.print(i);
       }
    System.out.print("\n");
    }
  System.out.println("Screen Display Done");
}
}
```

Example:

System.out.print(“Hello”);

System.out.print(“Java”);

Output:

Hello Java

The println( ) method, by contrast displays information followed by line feed.

Example:

System.out.println(“Hello”);

System.out.println(“Java”);

Output:

Hello

Java

4.10 STANDARD DEFAULT VALUES

In Java, every variable has a default value. If we don't initialize a variable when it is first created, Java provides default value to that variable type automatically as shown in following table.

Type of variable	Default Value
Byte	Zero : (byte)0
Short	Zero : (short)0
Int	Zero : 0
Long	Zero : 0L
Float	0.0f
Double	0.0d
Char	Null character
Boolean	False
Reference	null

Chapter 5 - Operators and Expressions

5.1 INTRODUCTION

Java supports a rich set of operators. We have already used several of them, such as =, +, - and *. An operator is a symbol that tells computer to perform certain mathematical or logical manipulations. They usually form a part of mathematical or logical expressions.

Java operators can be classified into a number of related categories as below:

1. Arithmetic Operators.
2. Relational Operators.
3. Logical Operators.
4. Assignment Operators.
5. Increment and Decrement Operators.
6. Conditional Operators.
7. Bitwise Operators.
8. Special Operators.

5.2 ARITHMATIC OPERATORS

Java provides all the basic arithmatic operators. These can operate on any built in numeric data type of Java. We cannot use these operators on Boolean type.

The list of operators is as follows:

Operators	Meaning
+	Addition or unary plus.
-	Subtraction or unary minus.
*	Multiplication.
/	Division.
%	Modulo division.

5.2.1 Integer Arithmetic

When the operands in a single arithmetic expression such as a+b are integers, the expression is called an *integer expression*, and the operation is called *integer arithmetic*. Integer arithmetic always yields an integer value. In the above example, if a and b are integers, then for a=14 and b=4 we the have the following results:

a + b = 18
a – b = 10
a * b = 56
a / b = 3 (decimal part truncated)
a % b = 2 (reminder of integer division)

a/b, when a and b are integer types, gives the result of division of a by b after truncating the divisor. This operation is integer division.

For modulo division, the sign of result is always the sign of first operand (the dividend). That is

-14 % 3 = 2
-14 % -3 = -2
14 % -3 = 2

(Note that modulo division is defined as: a%b = a-(a/b)*b, where a/b is integer division).

5.2.2 Real Arithmetic

An arithmetic operation involving only real operands is called *real arithmetic*. A real operand may assume values either in decimal or exponential notation. Since floating point values are rounded to the number of significant digits permissible, the final value is approximation of the correct result.

Modules operator % can be applied to floating point data as well. The floating-point module operator returns the floating-point equivalent of an integer division. What this means is that the division is carried out with both floating point operand, but the resulting divisor is treated as an integer, resulting in a floating point remainder.

```
class FloatPoint
{
   public static void main(String args[ ])
    {
        float a=20.5F, b=6.4F;
        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/b = " + (a/b));
        System.out.println("  a%b = " + (a%b));
     }
}
```

The output of Program is as follows:

a = 20.5

b = 6.4

a + b = 26.9

a – b = 14.1

a * b = 131.2

a / b = 3.20313

a % b = 1.3

5.2.3 Mixed – Mode Arithmetic

When one of the operands is real and the other is integer, the expression is called *mixed-mode arithmetic* expression. If either operand is of real type, then the other operand is converted to real and the real arithmetic is performed. The result will be real. Thus

15 / 10.0 produces the result 1.5

whereas

15 / 10 produces the result 1

More about mixed operations will be discussed when we deal with the evaluation of expressions.

5.3 RELATIONAL OPERATORS

We often compare two quantities, and depending on their relation, take certain decisions. These comparisons can be done with the help of relational operators. An expression such as

a < b or x < 20

Containing a relational operator is termed as a relational expression. The value of relational expression is either true or false. These operators and their meanings are as follows:

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

A simple relational expression contains only one relational operator is of the following form:

ae-1 relational operator ae-2

ae-1 and ae-2 are arithmetic expression, which may be simple constants, variables or combination of them.

5.3.1 Relational Expression

Expressions	Value
4.5 <= 10	True
4.5 < -10	False
-35 >=0	False

10 < 7 + 5	True
a + b == c + d	True

When certain arithmetic expressions are used on either side of relational operator, the arithmetic expressions will be evaluated first and then results compared. That is, arithmetic operators have higher priority over relational operators.

```
class RelationalOperators
{
   public static void main(String args[ ])
    {
       float a=15.0F, b=20.75F, c=15.0F;

       System.out.println("  a = " + a);
       System.out.println("  b = " + b);
       System.out.println("  c = " + c);

      System.out.println("  a < b is  " + (a<b));
      System.out.println("  a > b is  " + (a>b));
      System.out.println("  a == c is  " + (a==b));
      System.out.println("  a <= c is  " + (a<=c));
      System.out.println("  a >= b is  " + (a>=b));
      System.out.println("  b != c is  " + (b!=c));
      System.out.println("  b == a+c is  " + (b==a+c));
     }
}
```

The output of Program:

a = 15

b = 20.75

c = 15

a < b is true

a > b is false

a == c is true

a <= c is true

a >= b is false

b != c is true

b == a + c is false

5.4 LOGICAL OPERATORS:

Java has three logical operators.

Operator	Meaning
&&	Logical AND
||	Logical OR
!	Logical NOT

Like the simple relational expressions, a logical expression also yields a value of true or false, according to the truth table.

op-1	op-2	op-1 && op-2	op-1 || op-2
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Note:

- op-1 && op-2 is true if both op-1 and op-2 are true and false otherwise.
- op-1 || op-2 is false if both op-1 and op-2 are false and true otherwise.

5.5 ASSIGNMENT OPERATORS

Assignment operators are used to assign the value of an expression to a variable. We have seen the usual assignment operator, '='. In addition, Java has a set of 'shorthand' assignment operators, which are used in form

v op = exp;

where v is a variable, exp is an expression and op is a Java binary operator. The operator op = is known as shorthand assignment operator .

The assignment statement

v op = exp;

is equivalent to

v = v op(exp);

with v accessed only once. Consider an example

x += y+1;

This is same as the statement

x = x+(y+1);

Shorthand Assignment Operators

Statement with simple assignment operator	Statement with shorthand assignment operator
a = a + 1	a + = 1
a = a – 1	a -= 1
a = a * (n+1)	a * = n+1
a = a / (n+1)	a / = n+1
a = a % b	a % = b

The use of shorthand assignment operators has three advantages

- What appears on the left-hand side need not be repeated and therefore it becomes easier to write.
- The statement is more concise and easier to read.
- Use of shorthand operator results in a more efficient code.

5.6 INCEREMENT AND DECREMENT OPERATORS

Java has two very useful operators not generally found in many other languages. These are the increment and decrement operators (++ and --)

The operator ++ adds 1 to the operand while - - subtracts 1. Both are unary operators use in following form:

++m; or m++;

--m; or m--;

++m is equal to m = m + 1; (or m += 1)

--m is equal to m = m- 1; (or m -=1)

While ++m and m++ mean the same thing when they form statements independently, they behave differently when they are used in expressions on the right-hand side of an assignment statement. Consider following :

```
m = 5;
y = ++m;
```

In this case, the value of y and m would be 6. Suppose, if we rewrite the above statement as

```
m = 5;
y = m++;
```

then value of y would 5 and m would be 6.

```
class IncrementOperator
{
   public static void main(String args[])
    {
       int m=10, n=20;
       System.out.println("  m = " + m);
       System.out.println("  n = " + n);
       System.out.println("  ++m = " + ++m);
       System.out.println("  n++ = " + n++);
       System.out.println("  m = " + m);
       System.out.println("  n = " + n);
    }
}
```

Output of Program:

m = 10

n = 20

++m = 11

n++ = 20

m = 11

n = 21

5.7 CONDITIONAL OPERATOR

The character pair ? : is a ternary operator available in Java. This operator is used to construct conditional expression of the form

```
exp1 ? exp2 : exp3;
```

where exp1,exp2,exp3 are expression.

e.g.

```
a = 10;
b = 20;
x = (a > b ) ? a : b;
```

This is similar to say,

```
If (a > b)
    x = a;
else
    x = b;
```

5.8 BITWISE OPERATORS

Java supports bitwise operaors for manipulation of data at values of bit level. These operators are used for testing the bits, or shifting them to the right or left. Bitwise operators may not be applied to float or double.

Operator	Meaning
&	Bitwise AND
!	Bitwise OR
^	Bitwise exclusive OR
~	One’s compliment
<<	Shift left
>>	Shift right
>>>	Shift right with zero fill

5.9 SPECIAL OPERATORS

Java supports some special operators like instanceof operator and member selection operator(.).

5.9.1 instanceof Operator

The instanceof is an object reference operator and returns true if the object on the left-hand side is an instance of the class given on the right-hand side.

person instanceof student

Is true if the object person belongs to the class student; otherwise it is false.

5.9.2 Dot operator

The dot operator (.) is used to access the instace variables and methods of class objects.

e.g. person1.age, person1.salary()

5.10 PRECEDENCE OF ARITHMETIC OPERATORS

An arithmetic expression without any parentheses will be evaluated from left to right using the rules of precedence of operators. There are two distinct priority levels of arithmetic operators in Java

High priority * / %

Low priority + -

Chapter 6 - Decision making and Branching

A Java program is a set of statements, which are normally executed sequentially in the order in which they appear. This happens when options or repetitions of certain calculations are not necessary. However, in practice we have number of situations, where we may have to change the order of execution of statement based on certain condition. This involves decision-making. When a program breaks the sequential flow and jumps to another part of program, it is called branching.

Java language possesses such decision making capabilities and supports the following statements called as control or decision making statements.

1. if statement
2. switch statement
3. Conditional operator statement

6.1 DECISION MAKING WITH IF STATEMENT

The if statement is a powerful decision making statement and is used to control the flow of execution of statements. It is a two-way decision statement. It takes following form :

```
if (test expression)
```

It allows computer to evaluate test expression first and depending on the value of the expression is true or false, it transfers control to particular statement.

The if statement may be implemented into various form as follows:

1. Simple if statement
2. if…else statement
3. nested if…else statement
4. else if ladder

6.2 SIMPLE IF STATEMENT

The general form of a simple statement is

```
If (test expression)
{
        statement-block;
}
statement-x;
```

The statement-block may be single statement or a multiple statements. If test expression is true, the statement-block will be executed, otherwise statement – block will be skipped and execution will jump to statement-x.

Consider the following example:

```
If (category==sports)
{
marks = marks + bonus_marks;
}
System.out.println(marks);
```

In the above case expression category==sports becomes true then if block will be executed and bonus_marks will be added into marks otherwise statement after if block that is println statement will be executed.

6.3 THE IF….ELSE STATEMENT

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

```
if ( test exprssion)
{
    True-block statement(s)
}
else
{
     False-block statement(s)
}
statement-x
```

If the test expression is true, then the true-block statement(s) are executed; otherwise, the false-block statement(s) will be executed. In either case true or false block will be executed not both. In both cases, the control is transferred subsequently to the statement-x.

```
class IfElseTest
{
   public static void main(String args[])
    {
       int number=50;
       if(number % 2==0)
           {
                  System.out.println("Even Number");
           }
         else
           {
                  System.out.println("Odd Number");
           }
    }
}
```

6.4 NESTING OF IF….ELSE STATEMENTS

When a series of decisions are involved, we may have to use more than one if…else statement in nested form as follows.

```
if ( test condition1 )
{
    if ( test condition2 )
     {
        statement-1;
     }
    else
    {
```

```
        statement-2;
    }
}
else
{
        statement-3;
}
statement-x;
```

The logic of execution is that: If the condition1 is false , the statement3 will be executed; otherwise it continues to perform the second test. If condition2 is true , the statement-1 will be executed; otherwise the statemnt-2 will be evaluated and control transfer to statement-x.

```
class IfElseNesting
{
   public static void main(String args[])
    {
       int a=50 , b=100 ,c=75;
       if( a > b)   {
           if(a>c)   {
                System.out.println(a);
            }
            else  {
                 System.out.println(c);
            }
        }
       else    {
            if( c > b )  {
                 System.out.println(c);
             }
            else {
                 System.out.println(b);
            }
       }
    }
}
```

6.5 THE ELSE IF LADDER

There is another way of putting ifs together when multipath decisions are involved. It takes the following form:

```
if ( condition 1 )
        statement-1;
else if ( condition 2 )
        statement-2;
else if ( condition 3 )
        statement-3;
else if ( condition n )
        statement-n;
statement-x;
```

This construct is known as else if ladder. The condition are evaluated from the top, downwards. As soon as the true condition is found, the statement associated with it is executed and control is transferred to statement-x.

6.6 THE SWITCH STATEMENT

We have seen that when one many alternatives is to be selected, we can design program using if structure to control selection. However, complexity of such program increases dramatically as number of alternative increases.

Java has a built-in multiway decision statement known as switch. The switch tests value of expression against the list of case values and when a match is found, block statement associated with that case is executed. The general form of the switch statement is as follows:

```
switch(expression)
   {
      case value-1  :
                  block-1
                  break;
```

```
            case value-2  :
                        block-2
                        break;
                case value-n  :
                        block-n
                        break;
                default :
                        default-block
                        break;
        }
statement-x;
```

The expression is an integer expression or character. Value-1,value-2….. are constants and known as case labels. Each of these should unique within a switch statement. Block-1, block-2….. are statement lists and may contain zero or more statements. Each case label ends with colon (:).

When the switch is executed, the value of expression, is compared against the values value1,vallue2…. If case is found whose value matches with the value of expression, then block of statement that follows case are executed.

The break statement at the end of each block signals the end of particular case and causes exit from switch statement to statement-x.

The default is an optional case. When present, it will be executed if value of expression doesn't match any case values. If not present no action takes place when all matches fail and control goes to statement-x.

Chapter 7 - Classes , Objects and Methods

Java is a true object oriented programming language and therefore the underlying structure of all Java programs is classes. Anything we wish to represent in a Java program must be encapsulated in a class that defines the state and behavior of the basic program components known as objects. Classes create objects and objects use methods to communicate between them.

Classes provide a convenient method for packing together a group of logically related data items and functions that work on them. In Java data items are called fields and the functions are called methods. Calling a specific method in an object is described as sending the object a message.

7.1 DEFINING A CLASS

A class is user-defined data type with a template that serves to define its properties. Once the class type has been defined, we can create "variables" of that type using declarations similar to basic declarations. In Java, these variables are termed as instances of classes, which are the actual objects. The basic form of a class defination is :

```
class classname [ extends superclassname ]
{
        [variable declarations; ]
        [method declarations; ]
}
```

Everything inside the square bracket is optional. This means following would be valid class defination

```
class Empty
{
}
```

Because the body is empty, it can not do anything. However we con compile it and create object from it. The keyword ‘extends’ indicates that the properties of the

superclassname class are extended to the classname class. This concept is known as inheritance.

7.2 ADDING VARIABLES

Data is encapsulated in the class by placing data field inside the class body. These variables are called instance variables because they are created whenever an object of the class is initiated. We can define instance variables in same as local variables. Such as

```
class  Rectangle
{
    int length;
    int width;
}
```

The class rectangle contains two integer type instance variables. Remember these variables are only declared and therefore no storage space has been created in the memory. Instance variables are also known as member variables.

7.3 ADDING METHODS

A class with only data fields (without methods that operate on that data) has no life. The objects created by such class cannot respond to any messages. We must therefore add methods that are necessary for manipulating the data contained in the class. Methods are declared inside the body of the class but immediately after the declaration of instance variable. The general form of a method declaration is as follows:

```
type methodname ( parameter -list )
{
    method – body;
}
```

Method declarations have four basic parts:

- The name of the method (methodname)

- The type of the value the method returns (type)
- A list of parameters(parameter–list)
- The body of the method

```
class Rectangle
{
     int length,width;

     void getData(int x, int y)
       {
          length = x;
          width = y;
       }
     int rectArea( )
       {
          int area = length * width ;
          return( area ) ;
       }
}
```

Note that name of method is getdata . This method does not returning any value so return type is void. It accepts two arguments of type integer x and y.

7.4 CREATING OBJECTS

Objects in Java created using the ‘new’ operator. The new operator creates an object of specified class and returns a reference to that object. Here is an example of creating an object of previously created class Rectangle.

```
Rectangle rect1;
rect1= new Rectangle( );
```

The first statement declares a variable to hold object reference and second one actually assigns the object reference to the variable. The variable rect1 is now an object of Rectangle class.

Both statements can be combined into one as shown below:

```
Rectangle rect1 = new Rectangle();
```

The Rectangle( ) is default constructor for class.

7.5 ACCESSING CLASS MEMBERS

Now that we have created objects, each containing its own set of variables, we should assign values to these variables in order to use them in our program. All variables must be assigned values before they are used. Since we are outside the class, we cannot access the instance variables and methods directly. To do this, we must use the concerned object and dot operator as shown below:

objectname.variablename

objectname.methodname(parameter-list);

Here objectname is the name of the object, variablename is the name of the instance variable that we wish to access. methodname is name of the method that we wish to call, and parameter-list is a comma separated list of “actual values” that must match type and number of parameter list of the methodname declared in the class.

The instance variable of Rectangle class may be accessed and assigned values as follows:

rect1.length = 15;

rect1.width = 10;

The method getdata can be access to set the values of length and width as follows:

Rectangle rect1= new Rectangle();

rect1.getdata(15,10)

7.6 CONSTRUCTORS

We know that all the objects that are created must be given initial values. Java supports a special type of method, called constructor, which enables an object to initialize itself when it is created.

Constructors have same name as class name. Secondly, they do not specify return type, not even void. This is because they return the instance of the class itself.

```
class Rectangle
{
     int length,width;

     void Rectangle(int x, int y)
      {
         length = x;
         width = y;
      }

     int rectArea( )
     {
        int area = length * width ;
        return( area ) ;
     }
}

class ReactArea
{
      public static void main(String args[ ])
     {
      Rectangle rect1 = new Rectangle(15,10);
      int area1 = rect1.length * rect1.width;
      System.out.println("Area1 = " + area1);
     }
}
```

7.7 METHODS OVERLOADING

In Java, it is possible to create methods that have same name, but different parameter lists and different definations. This is called as method overloading. Method overloading is used when objects are required to perform similar tasks but using different input parameters. When we call a method in an object, Java matches up the method name first and then the number and type of parameters to decide which one of the definitions to execute. This process is called as polymorphism.

```
class Room
{
     float length;
     float breadth;

     Room (float a, float b)
     {
          length  = a;
          breadth = b;
     }
     Room (float a)
     {
          length  = a;
          breadth = a;
     }

     float area()
     {
          return (length * breadth);
     }
}
```

Here, we are overloading a constructor method room(). An object representing a rectangular room will be created as follows:

Room room1 = new room(12.0, 10.0);

On the other hand, if the room is square, then we may create corresponding object as

Room room2 = new Room(20.0);

7.8 STATIC MEMBERS

We have seen that a class basically contains two sections. One declares variables and other declares methods. These variables and method called instance variables and instance methods. This is because every time a class is initiated, a new copy of each of them is created.

Let us assume we want to define a member that is common to all the objects accessed without using particular object. Such member defined as follows:

static int count;

static int max (int x, int y);

The members that are declared as static are called static members.

Static variables are used when we want to have a variable common to all instances of a class.

```
class Mathoperation
{
  static int mul(int x,int y)
     {
        return x*y;
     }
  static int divide(int x,int y)
     {
        return x/y;
     }
}
class MathApplication
{
  public static void main(String args[ ])
     {
       int a = Mathoperation.mul(4,5);
       int b = Mathoperation.divide(a,5);
       System.out.println("b = " + b);
     }
}
```

Static methods have several restrictions.

1. They can only call other static methods.
2. They can only access static data.
3. They cannot refer to this or super in any way.

7.9 NESTING OF METHODS

We had seen that a method of a class, can be called only by an object of that class using dot operator. However there is exception to this. A method can be called by

using only its name by another method of same class. This is known as nesting methods.

In following Nesting class two methods are defined largest( ) and display( ). Display method calling largest method to find largest number and then display.

```
class Nesting
{
     int m,n;
     Nesting(int x, int y )
     {
        m = x;
        n = y;
     }
     int largest()
     {
       if( m >= n )
         return(m);
       else
         return(n);
     }
     void display()
     {
       int large = largest();
       System.out.println("Largest value = " +large);
     }
}
class NestingTest
{
     public static void main(String args[ ])
     {
          Nesting nest = new Nesting(50,40);
          nest.display();
     }
}
```

7.10 INHERITANCE : EXTENDING A CLASS

Reusability is an another feature of OOP. It is always nice if we reuse something that already exists rather than creating the same all over again. Java supports this

concept. Java classes can be reused in several ways. The mechanism of deriving a new class from an old one is called inheritance. The old class is called as base class or super class or parent class and the new class is called as subclass or derived class or child class.

The inheritance allows subclass to inherit all the variables and methods of their parent class. Inheritance may take different forms:

- Single inheritance (only one super class)
- Multiple inheritance (several super classes)
- Hierarchical inheritance (one super class , many subclasses)
- Multilevel Inheritance (Derived from derived class)

Java does not directly implement multiple inheritance. However this concept is implemented using secondary inheritance path in the form interfaces.

7.10.1 Defining a Subclass

A subclass is defined as follows:

```
class subclassname extends superclassname
{
        variables declarations;
        method declarations;
}
```

The keyword extends signifies that the properties of the superclassname are extended to subclassname.

The subclass will now contain its own variables and methods as well as those of the superclass. This kind of situation occurs when we want to add some more properties to an existing class without actually modifying it.

The output of program

```
Area1= 168
Volume1 = 1680
```

```
class Room
{
	int length;
	int breadth;
	Room(int a, int b)
	{
		length  = a;
		breadth= b;
	}
	int area()
	{
		return(length * breadth);
	}
}

class BedRoom extends Room
{
	int height;
	BedRoom(int x, int y, int z)
	{
		super(x,y);
		height=z;
	}
	int volume()
	{
		return(length * breadth * height);
	}
}

class InherTest
{
	public static void main(String args[])
	{
		BedRoom room1 = new BedRoom(14,12,10);
		int area1 = room1.area();
		int volume1 = room1.volume();
		System.out.println("Area = "+area1);
		System.out.println("Volume = "+volume1);
	}
}
```

The program defines a class Room and extends it to another class Bedroom. So Bedroom class includes 3 variables, namely, length, breadth and height and two methods area and volume.

The constructor in the derived class uses the super keyword to pass values that are required by the base constructor.

Finally, the object room1 of the subclass Bedroom calls the method area defined in the super class and method volume defined in subclass.

7.10.2 Subclass constructor

A subclass constructor is used to construct the instance variables of both the subclass and the superclass. The subclass constructor uses the keyword super to invoke the constructor method in superclass. The keyword super is used subject to the following conditions:

- super may only be used within a subclass constructor method.
- The call to superclass constructor must appear as the first statement within the subclass constructor.
- The parameters in the super call must match the order and type of instance variables declared in superclass.

7.10.3 Multilevel Inheritance

A common requirement in object-oriented programming is the use of a derived class as a super class. Java supports this concept and uses it extensively in building its class library. This concept allows us to build a chain of classes.

Grandfather	A	Superclass
Father	B	Intermediate superclass
Child	C	SubClass

The class A serves as base class for derived class B which in turn serves as base class the derived class C. The chain ABC is known as inheritance path.

Declaration as follows

```
class A
{
        ……..
        ……..
}
class B  extends A
{
        ……..
        ……..
}
class C extends  B
{
        ……..
        ……..
}
```

7.10.4 Hierarchical Inheritance

Another interesting application of inheritance is to use it as a support to hierarchical design of a program. Many programming problems can be cast into a hierarchy where many others below the level share certain features of one level.

7.11 OVERRIDING METHODS

We have seen that a method defined in a super class is inherited by its subclass and is used by the objects created by the subclass. Method inheritance enables us to define and use methods repeatedly in subclass without having to define the methods in subclass.

However, there may be occasions when we want an object to respond to the same method but have different behavior when that method is called. That means, we should override method defined in the superclass. This is possible to define method that having same name, same arguments and return type in the superclass

. Then, when that method is called method defined in subclass is invoked and executed instead of one in the super class. This is known as overriding.

```
class Super
{
int x;
Super(int a)
{
x = a;
}
void display()
{
System.out.println("Super x = "+x);
}
}

class Sub extends Super
{
int y;
Sub(int a, int b)
{
super(a);
y=b;
}
void display()
{
System.out.println("Super x = "+x);
System.out.println("Sub y = "+y);
}
}
class OverrideTest
{
public static void main(String args[])
{
Sub s1 = new Sub(100,200);
s1.display();
}
}
```

7.12 FINAL VARIABLES AND METHODS

All methods and variables can be overridden by default in subclass. If we wish to prevent the subclass from overriding the members of the superclass, we can declare them as final using the keyword final as a modifier.

```
final int size = 100;
final void showstatus( ) { …….. }
```

7.13 FINAL CLASSES

Sometimes we may like to prevent class being further subclassed for security reasons. A class that can not be subclassed is called a final class. This is achieved in Java using keyword final as follows:

```
final class Aclass { …….. }
final class Bclass extends Someclass { …… }
```

Any attempt to inherit these classes will cause an error and compiler will not allow it.

Declaring a class final prevents any unwanted extensions to class.

7.14 FINALIZER METHOD

We have seen that a constructor method is used to initialize an object when it is declared. This process is known as initialization. Similarly Java supports concepts called finalization, which is just opposite to initialization. Java runtime is an automatically garbage collection system. This means that it automatically frees up memory resources used by the objects. But objects may hold other non object resources such as a file descriptor or window system fonts. The garbage collector can not free these resources. In order to free these resources we must use a finalizer method. This is similar to destructors in c++.

The finalizer method is simply finalize() and can be added to any class. Java calls that method whenever it is about to reclaim the space for that object. The finalize method should explicitly define the tasks to be performed.

7.15 ABSTRACT METHOD AND CLASSES

We have seen that by making method final we ensure that the method is not redefined in a subclass. That is, the method never be subclassed. Java allows us to do something that is exactly opposite to this. That is, we can indicate that method must always be redefined in a subclass, thus making overriding compulsory. This is done using modifier keyword abstract in the method defination.

```
abstract   class Shape
{
     ……..
     ……..
   abstract void draw ( );
    ……
    ……
}
```

When a class contains one or more abstract methods, it should also be declared abstract as shown above.

While using abstract classes, we must satisfy the following conditions

- We can not use abstract classes to instantiate object directly. for example
  Shape s = new Shape( ) ;
  is illigal because shape is an abstract class.
- The abstract methods of an abstract class must be defined in its subclass.
- We can not declare abstract constructors or abstract static methods.

7.16 VISIBILITY CONTROL

All the members of a class can be inherited by subclass with the help of keyword extends. Variables and methods of a class are visible everywhere in the program. However, it may be necessary in some situations to restrict the access to certain variables and methods from outside the class. For example, we may not like the objects of a class directly alter the value of a variable or access method. We can

achieve this in Java by applying visibility modifiers to the instance variables and methods. The visibility modifiers also known as access modifiers. Java provides three types of access modifiers: public, private and protected. They provide different levels of protection as follows:

7.16.1 Public Access

Any variable or method is visible to the entire class in which it defined. What if we want to make it visible to all classes outside this class? This is possible by simply declaring the variable or method public.

```
public int number;
public void sum ( ) { …… }
```

A variable or method declared as public has the widest possible visibility and accessible everywhere.

7.16.2 Friendly Access

When no access modifier is specified then the members defaults to a limited version of public accessibility known as “friendly” level of access.

The difference between “public” access and “friendly” access is that public modifier makes fields visible in all classes, regardless of their packages while the friendly access makes fields visible only in same package, but not in other packages.

7.16.3 Protected Access

The visibility level of a “protected” field lies in between public access and friendly access. That is, protected modifier makes the fields visible not only to all classes and subclasses of one package but also to subclass in other packages. Note that non-subclasses in other packages can not access the protected members.

7.16.4 Private Access

Private fields enjoys the highest degree of protection. They are accessible only within their own class. They can not be inherited by subclass and therefore not accessible in subclasses. A method declared as private behaves like method

declared as final. It prevents the method from being subclassed. Also note that we cannot override a non-private method in a subclass and then make it private.

7.16.5 Private Protected Access

A can be declared with two keywords private and protected together like

```
private protected int number;
```

Not supported in new versions.

This gives visibility level between private and protected. This modifier makes fields visible in all subclasses regardless of what package they are in. Remember these fields are not accessible by other classes in the same package. Following table summarizes the visibility provided by various access modifiers.

Access Modifier → / Access *Location* ↓	Public	Protected	Friendly (default)	Private
Same class	Yes	Yes	Yes	Yes
Subclass in same package	Yes	Yes	Yes	No
Other classes in same package	Yes	Yes	Yes	No
Subclass in other packages	Yes	Yes	No	No
Non subclasses in other packages	Yes	No	No	No

Chapter 8 - Exception Handling

8.1 INTRODUCTION

Rarely does a program run successfully at its very first attempt. It is common to make mistakes while developing as well as typing a program. A mistake might lead to an error causing the program to produce unexpected results. Errors are the wrongs that can make a program go wrong.

An error may produce an incorrect output or may terminate the execution of the program abruptly or may cause the system to crash. It is therefore important to detect and manage properly all the possible error conditions in the program so that the program will not terminate or crash during execution.

8.2 TYPES OF ERRORS

Errors may broadly be classified into two categories:

- Compile-time errors
- Run-time errors

8.2.1 Compile-Time Errors

All syntax errors will be detected and displayed by the Java compiler and therefore these errors are known as compile-time errors. Whenever the compiler displays an error, it will not create the **.class** file. It is therefore necessary that we fix all the errors before we successfully compile and run program.

8.2.2 Run-Time Errors

Sometimes, a program may compile successfully creating the **.class** file but may not run properly. Such programs may produce wrong results due to wrong logic or may terminate due to errors such as stack overflow. Most common run-time errors are:

Dividing an integer by zero

Accessing an element that is out of bounds of array

Trying to store a value into an array of an incompatible class or type

Passing a parameter that is not in a valid range or value for a method
Trying to illegally change the state for array
Using a null object reference as a legitimate object reference to access a method or a variable
Converting invalid string to a number
Accessing a character that is out of bounds of a string
And many more
When such errors are encountered, Java typically generates an error message and aborts the program. Program 5.1 illustrates how a run-time error causes termination of execution of the program.

Program 5.1 : To illustrate run-time errors

```
class Error 1
{
public static void main(string args[ ])
{
int a= 10;
int b= 5;
int c= 5;

int x= a/(b-c);                    // Division by zero
System.out.println("x = " +x);
int y =  a/(b+c);
System.out.println("y = " + y);
}
}
```

The Program above is syntactically correct and therefore does not has any problem during compilation. However, while executing, it displays the following message and stops without executing further statements.

```
java.lang.ArithmeticException :  / by zero
at Error2.main (Error2.java:10)
```

When Java run-time tries to execute a division by zero, it generates an error condition, which causes the program to stop after displaying an appropriate message.

8.3 EXCEPTIONS

An exception in Java is a signal that indicates the occurrence of some unexpected condition during execution. Java provides an exception handling mechanism for systematically dealing with such error conditions. The purpose of exception handling mechanism is to provide a means to detect and report an "exceptional circumstance" so that appropriate action can be taken. The mechanism suggests incorporation of a separate error handling code that performs the following tasks:

Find the problem (**Hit** the exception).
Inform that an error has occurred (**throw** the exception)
Receive the error information (**catch** the execption)
Take corrective actions (**Handle** the exception)

The error handling code basically consists of two segments, one to "throw" exceptions and the other to "catch" exceptions and to take appropriate actions When writing programs, we must always be on the lookout for places in the program where an exception could be generated. Some common exceptions that we must watch out for catching are listed in the following table.

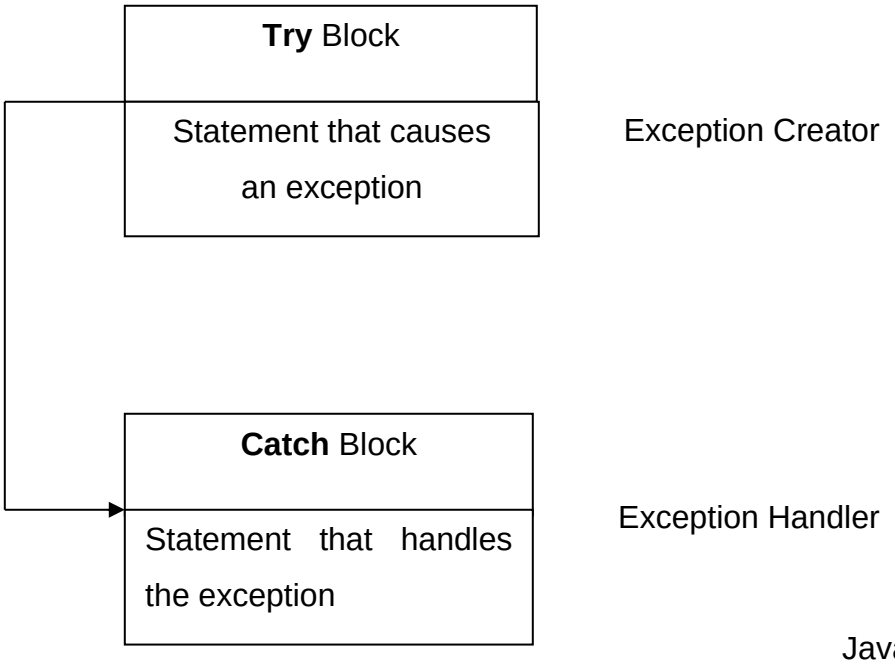
Common Java Exceptions

Exception Type	Cause of Exception
ArithmeticException	Caused by maths errors such as division by zero
Array IndexOutof BoundsException	Caused by bad arrays indexes

ArraysStoreException	Caused when a program tries to stores the wrong type of data in an array
FileNotFoundException	Caused by an attempt to access a nonexistent file
IOException	Caused by general I/O failures, such as inability to read from a file
NullPointerException	Caused by referencing a null object
NumberFormatException	Caused when a conversion between strings and number fails
OutOfMemoryException	Caused when there's not enough memory to allocate a new object
SecurityException	Caused when an applet tries to perform an action not allowed by browser's security setting
StackOverflowException	Caused when the system runs out of stack space
StringIndexOutofBounds Exception	Caused when a program attempts to access a nonexistent character position in a string

8.4 SYNTAX OF EXCEPTION HANDLING CODE

The basic concepts of exception handling are throwing an exception and catching it. This is illustrated in the following figure.



Java uses keyword **try** to preface a block of code that is likely to cause an error condition and " throw" an exception. A catch block defined by the keyword **catch** "catches" the exception "thrown" by the try block and handles it appropriately. The catch block is added immediately after the try block. The following example illustrates the use of simple **try** and **catch** statements.

```
try
{
        statements;             //generates exception
}
catch(Exception-type e)
{
        statements;              //handles exception
}
```

The try block can have one or more statements that could generate an exception. If any one statement generates an exception, the remaining statements in the block are skipped and execution jumps to the catch block that is placed next to the try block.

The catch block too can have one or more statements that are necessary to process the exception, remember that every **try** statement should be followed by at least one catch statement; otherwise compilation error will occur.

Note that the **catch** statement should work like a method definition. The **catch** statement is passed as a single parameter, which is reference to the exception object thrown (by the try block). If the catch parameter matches with the type of exception object thrown, then the exception is caught and statements in the catch block will be executed. Otherwise, the exception is not caught and the default exception handler will cause the execution to terminate.

Class Error3

```
{
        public static void main (String args [ ])
        {
                int a =  10;
                int b =  5;
                int c =  5;
                int x, y ;
                try
                {
                        int x= a/(b-c);   // Exception here
                }
                catch(ArithmeticException e)
                {
                        System.out.println("Division by zero" );
                }

                int y =  a/(b+c);
                System.out.println("y = " + y);
        }
}
```

Output of the program 5.2 is:

division by zero

y = 1

Note that the program did not stop at the point of exceptional condition. It catches the error condition, prints the error message, and then continues the execution, as if nothing has happened. Compare with the output of Program 5.1 which did not give the value of y.

Chapter 9 - ARRAYS, STRINGS AND VECTORS

9.1 ARRAYS

An array is a group of contiguous or related data items that share a common name. For instance, we can define an array name salary to represent a set of salaries of group of employees particular number of employees are specified by specifying array size which is enclose between square brackets. For example:

salary[10]

Represents salary of 10 employees. While complete set of values is referred to as an array, the individual values are called elements.

9.2 ONE-DIMENSIONAL ARRAYS

A list of items can be given one variable name using only one subscript and such a variable is called single-subscripted variable or a one-dimensional array.

9.3 CREATING AN ARRAY

Like any other variables, array must be declared and created in computer memory before they are used. Creation of array involves three steps:

1. Declare the array.
2. Create memory locations
3. Put values into memory locations.

9.3.1 Declaration of Arrays

Arrays in Java can be defined in two forms:

```
type arrayname[ ];
type[ ] arrayname;
```

Examples:

```
int number[ ];
float average[ ];
int[ ] counter;
float[ ]  marks;
```

Remember that we do not enter the size of the arrays in declaration.

9.3.2 Creation Of Arrays

After declaring an array, we need to create it in memory. Java allows us to create array using new operator only, as shown below

arrayname = new type[size];

Examples:

number = new int[5];

average = new float[10];

9.3.3 Initialization Of Arrays

The final step is to put values into the array created. This process is known as initialization. This is done using the array subscript as follows:

Example:

number[0] = 35;

number[1] = 40;

……..

…….

number[4]=19;

Note that Java creates arrays starting with a subscript of 0 and ends with a value less than the specified size.

Unlike in C, Java protects arrays from overruns and underruns. Trying to access an array beyond its boundaries will generate an error message.

We can also initialize arrays automatically in the same way as ordinary variables when they are declared as shown below:

type arrayname[ ] = { list of values };

The array initializer is a list of values separated by commas and surrounded by curly braces. Note that no size is given. The compiler allocates space for all elements automatically.

Example:

int number[ ] = {35,40,20,57,19};

It is possible to assign an array object to another.

int a[ ] = {1,2,3};

int b[ ] ;

b = a;

Are valid in Java. Both the arrays will have same values.

9.3.4 Array Length

In Java, all arrays store the allocated size in a variable named length. We can access the length of array a using a.length. Example:

int asize = a.length;

This informaton will be useful in manipulation of arrays when their sizes are not known.

```
class Numbersorting
{
public static void main(String args[ ])
{
     int no[]={55,78,34,67,25};
     int n=no.length;
     System.out.println("Given List is:  " );
     for(int i=0; i<n; i++)
     {
          for(int j=i+1;j<n;j++)
          {
               if(no[i] < no[j])
               {
               int temp=no[i];
               no[i]=no[j];
               no[j]=temp;
               }
          }
     }
     System.out.println("Sorted List : " );
     for(int i=0; i<n; i++)
     {
     System.out.println(" "   +no[i]);
     }
}
}
```

9.4 STRINGS

String manipulation is the most common part of many Java programs. Strings represent sequence of characters. The easiest way of representing sequence of characters in Java is by using a character array.

```
char charArray [ ] = new char[4];
charArray[0] = 'J';
charArray[1] = 'a';
charArray[2] = 'v';
charArray[3] = 'a';
```

In Java, strings are class objects and implemented using two classes, namely, String and StringBuffer. A Java string is an instantiated object of the String class. A Java string is not a character array and is not a NULL terminated. String may declared and created as follows:.

```
String stringName;
stringName = new String("string");
```

Example:

```
String firstname;
firstName = new String("Anil");
```

These two statements may be combined as follows:

```
String firstName = new String("Anil");
```

Like array it is possible to get the length of string using the length method of String class.

```
int m = firstName.length( );
```

Note the use of parentheses here.

9.5 STRING ARRAYS

We can also create and use arrays that contain strings. The statement

```
String itemArray[] = new String[3];
```

Will create an itemArray of size 3 to hold three string constants.

9.5.1 String Methods

The String class defines a number of methods as allow us to accomplish a variety of string manipulation tasks. Following table shows some commonly used string methods, and their tasks.

Method Call	**Task performed**
S2 = s1.toLowerCase	Converts the string s1 to all lowercase
S2 = s1.toUpperCase	Converts the string s1 to all Uppercase
S2 = s1.replace('x','y')	Relaces all appearance of x with y
S2 = s1.trim()	Remove white spacesat the beginning and end of the string s1
S1.equals(s2)	Returns true if s1 is equal to s2
S1.equalsIgnoreCase(s2)	Returns true if s1 = s2, ignoring case
S1.length()	Gives the length of s1
S1.CharAt(n)	Gives nth character of s1
S1.compareTo(s2)	Gives –ve if s1<s2, +ve if s1>s2 and 0 if s1=s2
S1.concat(s2)	Concatenates s1 and s2
S1.substring(n)	Gives substring starting from nth character
S1.substring(n,m)	Gives substring starting from n^{th} character up to m^{th} (not including mnot including m^{th})
String.ValueOf(p)	Creates a string object of the parameter p
p.toString()	Creates a string representation of the object p
S1.indexOf('X')	Gives the position of the first occurrence of 'x' in the string s1
S1.indexOf('x',n)	Gives the position of 'x' that occurs after nth position in the string s1
String.ValueOf(variable)	Converts the parameter value to string representation

9.5.2 StringBuffer Class

StringBuffer is a peer class of string. While String creates strings of fixed_length, StringBuffer creates strings of flexible length that can be modified in terms of both length and content. We can insert characters and substrings in the middle of a string, or append another string to the end. Table shows some of methods and their use.

Method	Task
S1.setCharAt(n,'x')	Modifies the nth character to x
S1.append(s2)	Appends the string s2 to s1 at the end
S1.insert(n,s2)	Inserts the string s2 at the position n of the string s1
S1.setLength(n)	Sets the length of the string s1 to n. if n < s1.length(), s1 is truncated. Else zeros are added to s1.

9.6 VECTORS

C and C++ programmers will know that generic utility functions with variable arguments can be used to pass different arguments depending upon calling situations. Java does not support the concept of variable arguments to a function. This feature can be achieved in Java through the use of the Vector class contained in java.util package. This class can be used to create a generic dynamic array known as vector that can hold object of any type and any number. The objects do not have to be homogeneous. Arrays can be easily implemented as vectors. Vectors are created as follows:

```
Vector intVect = new Vector();
Vector list = new Vector(3);
```

Note that a vector can be declared without specifying any size explicitly. A vector can accommodate an unknown number of items. Even, when a size is specified, this can be overlooked and a different number of items may be put into the vector. Remember, in contrast, an array must always have its size specified.

Vector possess a number of advantages over arrays.

1. It is convenient to use vectors to store objects.

2. A vector can be used to store list of objects that may vary in size.
3. We can add and delete objects from the list as and when required.

A major constraint in using vector is that we can not directly store simple data types in a vector; we can only store objects. Therefore, we need to convert simple types to objects. This can be done with the help of wrapper class. The vector class supports a number of methods that can be used to manipulate the vectors created. Important ones are listed below.

Method call	Tasks performed
list.addElement(item)	Adds the item specified to the list at the end
list.elementAt(10)	Gives the name of the 10th object
list.size( )	Gives number of objects present
list.removeElement(item)	Removes the specified item from the list
list.removeElementAt(n)	Removes the item stored in the nth position of the list
list.copyInto(array)	Copies all the items from list to array
list.insertElement (item,n)	Inserts the item at the nth positon

9.7 WRAPPER CLASS

As pointed out earlier, vectors can cot handle primitive data types like int, float, long, char and double. Primitive data types must be converted into object types by using wrapper classes contained in java.lang package. Table shows the simple data types and their corresponding wrapper class types.

Simple Type	**Wrapper Class**
boolean	Boolean
char	Character
double	Double
float	Float
int	Integer
long	Long

Chapter 10 - INTERFACES: MULTIPLE INHERITANCE

In classes and objects's chapter, we discussed about classes and how they can be inherited by other classes. We also learned about various forms of inheritance and pointed out that Java does not support multiple inheritance. That is, classes in Java can not have more than one superclass. For instance defination like

```
Class A extends b extends c
{
        ……….
        ……….
}
```

is not permitted in Java. However, Java designers could not overlook the importance of multiple inheritance. A large number of real life problems require the use of multiple inheritance. Since C++ like implementation of inheritance proves difficult and more complexity to the language, Java provides an alternate approach known as interfaces to support the concept of multiple inheritance. Although a Java class can not be a subclass of more than one superclass, it can implement more than one interface, thereby enabling us to create classes that build upon other classes without the problems created by multiple inheritance.

10.1 DEFINING INTERFACES

An interface is basically a kind of class. Like classes, interfaces contain methods and variables but with major difference. The difference is that interfaces define only abstract methods and final fields. This means that interfaces do not specify any code to implement these methods and data fields contain only constants.

Therefore, it is the responsibility of the class that implements an interface to define the code for implementation of these methods.

The syntax of defining interface is very similar to defining a class. The general form is:

```
interface InterfaceName
{
   variables declaration;
   methods declaration;
}
```

here, interface is the keyword and InterfaceName is any valid Java variable. Variable are declared as follows:

```
   static final type variableName = value;
```

note that variables are declared as constants. Methods declaration will contain only a list of methods without any body statements. Example:

```
 return-type methodName1( parameter_list);
```

Here is an example of an interface defination that contains two variables and one method.

```
interface Item
{
      static final int code = 1001;
      static final String name = “Fands”;
      void display( );
}
```

Note that the code for the method is not included in interface and method declaration simply ends with semicolon. The class that implements this interface must define the code for the method.

10.2 EXTENDING INTERFACES

Like classes, interface can also be extended. That is, an interface can be subinterfaced from other interfaces. The new subinterface will inherit all the

members of the superinterface in manner similar to subclasses. This is achieved using keyword extends as follows:

```
interface name2 extends name1
{
        body of name2
}
```

For example we can put all constants in one interface and the methods in the other. This allows to use the constants in classes where methods are not required.

```
interface ItemConstants
{
        int code = 1001;
        string name = “fands”;
}
interface Item extends ItemConstants
{
    void display( );
}
```

The interface Item would inherit the constants, code and name into it. Note that variable name and code are declared like simple variables. It is allowed because all the variables in an interface are treated as constants although the keywords final or static are not present.

We also combine several interfaces together into single interface. As follows:

```
interface ItemConstants
{
        int code = 1001;
        string name = “fands”;
}
interface ItemMethods
 {
    void display( );
```

```
}
interface  Item extends ItemConstants, ItemMethods
{
      ……..
      ……..
}
```

While interfaces are allowed to extend to other interfaces, subinterfaces can not define the methods declared in the superinterfaces. After all subinterfaces are still interfaces, not classes. Instead, it is the resposibility of any class that implements the derived interface to define all the methods. Note that when an interface extends two or more interfaces, they are separated by commas.

10.3 IMPLEMENTIG INTERFACES

Interfaces are used as “superclasses” whose properties are inherited by classes. It is therefore necessary to create a class that inherits the given interface. This is done as follows:

```
class classname implements Interfacename
{
    body of classname
}
```

Here the class classname implements the interface interfacename. A more general form of implementation may look like this:

```
   class classname extend superclass
            implements interface1, interface2,…….
{
    body of classname
}
```

This shows that class can extend another class while implementing interfaces.

When class implements more than one interface, a comma separates them. Implementation of interfaces as class types is shown in program:

```
interface Area
{
  final static float pi=3.14F;
  float compute(float x,float y);
}
class Rectangle implements Area
{
  public float compute(float x, float y)
        {
          return(x*y);
        }
}

class Circle implements Area
{
  public float compute(float x, float y)
        {
          return(pi*x*y);
        }
}

class InterfaceTest
{
  public static void main(String args[ ])
        {
        Rectangle rect=new Rectangle( );
        Circle cir=new Circle();
        Area area;
        area = rect;
        System.out.println("Area of Rectangle = " +area.compute(10,20));
        area = cir;
        System.out.println("Area of Circle = " +area.compute(10,0));
         }
}
```

In this program we have created an interface Area and implement the same in 2 different classes, Rectangle and Circle. We create an instance of each class using the new operator. Then we declare an object of type area, the interface class. Now we assign the reference to Rectangle object rect to area. When we call compute

method of area, the compute method of Rectangle class is invoked. The same thing is repeated with the circle object.

10.4 ACCESSING INTERFACE VARIABLES

Interfaces can be used to declare a set of constants that can be used in different classes. This is similar to creating header files in C++ to contain large number of constants. The constant value will be available to any class that implements the interface.

Following program illustrates the implementation of the concept of multiple inheritance using interfaces.

```
class Student {
          int rn;
          void getNumber(int n) {
                    rn = n;
         }
          void putNumber() {
            System.out.println("Rollnum = " +rn);
         }
}
class Test extends Student
{
         float p1,p2;
         void getMarks(float m1, float m2) {
            p1 = m1;
            p2 = m2;
         }
         void putMarks() {
                  System.out.println("Mark1  = " +p1);
                  System.out.println("Mark = " +p2);
         }
}
interface Sports
         {
         float sportWt = 6.40F;
         void putWt();
         }
```

```
class Results extends Test implements Sports
{
         float tot;
         public void putWt()
         {
```

Chapter 11 - AWT

11.1 ABSTRACT WINDOW TOOLKIT

This package contains many classes, which will allow us to create good graphical user interface.

We will have a list of classes and corresponding methods below. But the best method to understand what a class is doing, we will have code examples and corresponding screen layout at the end of this chapter.

11.2 TEXTFIELD

addActionListener	adds ActionListener to the text field to receive action events.
echoCharIsSet()	returns true if the text field has echoing.
GetText()	gets the text from keyboard.
SetText	sets the text to specified string.

11.3 TEXTAREA

TextArea( )	constructs a new text area.
TextArea(int,int)	creates a new empty text area with specified number rows and columns.
TextArea(string)	constructs a new text area with specified text.
TextArea(string,int,int)	constructs a new text area with specified text and number of rows and columns.
append(string)	appends given text to end of the text area's current text.
getRows( )	returns number of rows in the text area.
getColumns( )	returns number of columns in the text area.

replaceRange(string,int,int)	replaces text from the indicated start to end position with the new text given.
setColumns(int)	sets number of columns for this text area.
setRows(int)	sets number of rows for this text area.

11.4 BUTTON

Button( )	constructs a button with no label.
Button(string)	constructs a button with a given label.
AddActionListener(ActionListener)	adds the ActionListener to the button to receive action event.
getLabel( )	gets the label of a botton.
setLabel( )	sets the button's label.

11.5 LABEL

Label( )	constructs an empty label.
Label(String)	constructs a label with specified string.
Label(String,int)	constructs a label with specified string of text and alignment
getAlignment( )	gets the current alignment of the label.
getText( )	get the text of label.
setAlignment(int)	sets the current alignment for the label.
setText(String)	sets the text for this label to specified string.

11.6 CHECKBOX

Checkbox( )	constructs a check box with no label.
Checkbox(String)	constructs a check box with given label.

Checkbox(String,boolean)	constructs a check box with given label and sets the specified boolean state.
Checkbox(String,boolean.ChechboxGroup)	constructs a check box with given label, set to the given boolean state , in the specified check box group.
Checkbox(String,ChechboxGroup,boolean)	constructs a check box with given label, set to the specified boolean state , in the specified check box group.
addItemListener(ItemListener)	adds the given Item Listener to receive item events from check box.
getCheckboxGroup( )	returns the check box group for this check box.
getLabel( )	get the check box's label.
getState( )	returns the boolean state of the check box to determine whether this check box is "on" or "of" state.
setLabel(String)	sets this check box label to given string.
setState(Boolean)	sets this check box state.

11.6.1 CheckboxGroup (For radio button implementation)

ChechboxGroup( )	Creates a check box group.
getSelectedCheckbox ( )	Get the current selected checkbox from check box group.
setSelectedCheckbox (Checkbox)	Set the current choice in this check box's group to specified checkbox.

11.7 SCROLLBAR

Scrollbar( )	Constructs a new vertical scroll bar.

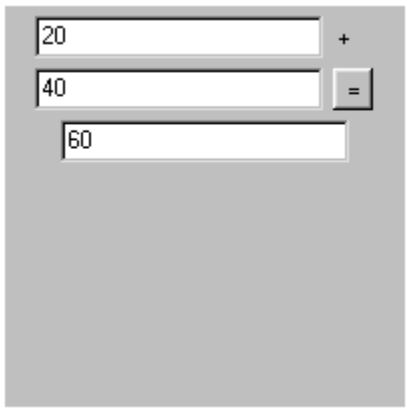
Scrollbar(int)	Constructs a new vertical scroll bar wit specified orientation.
Scrollbar(int,int,int,int)	Construct a new vertical scroll bar wit specified orientation, initial, value, scroll thumb size, minimum and maximum values.
getMaximum( )	Get maximum setting of this scrollbar.
getMaximum( )	Get maximum setting of this scrollbar.
getBlockIncrement( )	Get block increment of this scrollbar.
getMinimum( )	Get maximum setting of this scrollbar.
getOrientation( )	Get orientation setting of this scrollbar.
getValue( )	Get the current value of this scrollbar.
setBlocIncrement(int)	Set maximum setting for this scrollbar.
setMaximum(int)	Set maximum setting for this scrollbar.
setMinimum(int)	Set minimum setting for this scrollbar.
setOrientation(int)	Set block increment of this scrollbar
setValue(int)	Set the current value of this scrollbar.
setValues(int)	Set the values of this scrollbar.

11.8 AWT EXAMPLES

11.8.1 SIMPLE ADDER

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


public class adder extends Applet implements ActionListener
{

TextField text1, text2, anstext;
Button b1;
```

```
Label L1;

public void init()
{
 text1 = new TextField(20);
 add(text1);
 L1 = new Label("+");
 add(L1);
 text2 = new TextField(20);
 add(text2);
 b1 = new Button("=");
 add(b1);
 b1.addActionListener(this);
 anstext = new TextField(20);
 add(anstext);
}

public void actionPerformed(ActionEvent e)
{
if (e.getSource() == b1)
{
int       sum       =       Integer.parseInt(text1.getText())
+Integer.parseInt(text2.getText());
anstext.setText(String.valueOf(sum));
}
}
}
```



11.8.2 CLICKER EXAMPLE

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

public class clicker extends Applet implements ActionListener
{
     // Declare a text field
     TextField text1;
     Button button1;

     public void init()
     {
          //Create a text field
          text1 = new TextField(20);
          add(text1);

          button1 = new Button("OK");
          add(button1);
          button1.addActionListener(this);
     }
          public void actionPerformed(ActionEvent event)
     {
          String msg = new String ("Welcome to Java");
          if(event.getSource()==button1)
     {
// Place a text in text field
text1.setText(msg);

     }
     }
}
```

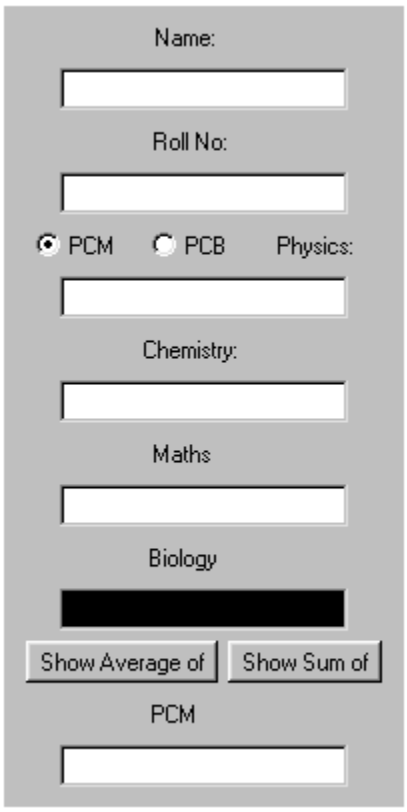
11.8.3 RADIO GROUPS

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.awt.Component.*;

public class enabledisable extends Applet implements
ItemListener, ActionListener
{
 Label L1, L2, L3, L4, L5, L6, L7;
 TextField T1, T2, T3, T4, T5, T6, T7;
 CheckboxGroup CG1;
 Checkbox C1, C2,  C3;
 Button B1, B2;
 int x;
public void init()
{
 L1 = new Label("Name:   ");
 add(L1);

 T1 = new TextField(20);
 add(T1);

 L2 = new Label("Roll No: ");
 add(L2);

 T2 = new TextField(20);
 add(T2);

 CG1 = new CheckboxGroup();

 C1 = new Checkbox("PCM" ,  true, CG1);
 add(C1);
 C1.addItemListener(this);

 C2 = new Checkbox("PCB" ,  false, CG1);
 add(C2);
 C2.addItemListener(this);

 L3 = new Label("Physics: ");
```

```
 add(L3);
 L3.setAlignment(Label.CENTER);

 T3 = new TextField(20);
 add(T3);

 L4 = new Label("Chemistry: ");
 add(L4);

 T4 = new TextField(20);
 add(T4);

 L5 = new Label("Maths    ");
 add(L5);

 T5 = new TextField(20);
 add(T5);

// C3 = new Checkbox("Show Average of");
//add(C3);
// C3.addItemListener(this);

 L7 = new Label("Biology    ");
 add(L7);

 T7 = new TextField(20);
 add(T7);


B1 = new Button("Show Average of");
add(B1);
B1.addActionListener(this);
     B2 = new Button("Show Sum of");
     add(B2);
     B2.addActionListener(this);
 L6 = new Label("PCM       ");
 add(L6);

 T6 = new TextField(20);
 add(T6);

x = 1;
```

```
}
 public void itemStateChanged(ItemEvent e)
{
 if (e.getItemSelectable() == C1)
{
// L5.setText("Maths");
 T7.setEnabled(false);
 T5.setEnabled(true);
 L6.setText("PCM");
 x =1;
 repaint();
}
else {
// L5.setText("Biology");
 L6.setText("PCB");
 T5.setEnabled(false);
 T7.setEnabled(true);
 x= 2;
 repaint();
}
}
 public void actionPerformed(ActionEvent e)
{
 if (e.getSource() == B1 )
{
 if ( x == 1)
{
int s2 = Integer.parseInt(T3.getText());
int s3 = Integer.parseInt(T4.getText());
int s4 = Integer.parseInt(T5.getText());
int s5 = (s2+ s3+ s4)/3;
T6.setText(String.valueOf(s5));
}
else
 {
int s2 = Integer.parseInt(T3.getText());
int s3 = Integer.parseInt(T4.getText());
int s4 = Integer.parseInt(T7.getText());
int s5 = (s2+ s3+ s4)/3;
T6.setText(String.valueOf(s5));
}
}
```

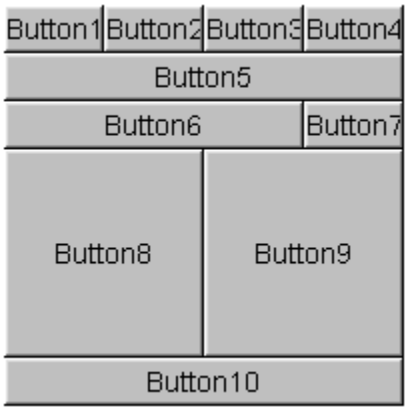
```
 if (e.getSource() ==B2)
{
 if ( x == 1)
{
int s2 = Integer.parseInt(T3.getText());
int s3 = Integer.parseInt(T4.getText());
int s4 = Integer.parseInt(T5.getText());
int s5 = (s2+ s3+ s4);
T6.setText(String.valueOf(s5));
}
else
 {
int s2 = Integer.parseInt(T3.getText());
int s3 = Integer.parseInt(T4.getText());
int s4 = Integer.parseInt(T7.getText());
int s5 = (s2+ s3+ s4);
T6.setText(String.valueOf(s5));
}
}
}

 public void paint(Graphics g)
{
  if ( x == 1)
{
 T7.setBackground(Color.black);
}else{
T7.setBackground(Color.white);
}
 if (x == 2)
{
 T5.setBackground(Color.black);
}else{
T5.setBackground(Color.white);
}
}
}
```



11.8.4 GRID BAG EXAMPLE

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

public class gridbagexample extends Applet
{

public void init()
{
 GridBagLayout gridbag = new GridBagLayout();
 GridBagConstraints c = new GridBagConstraints();

 setFont(new Font("Helvetica", Font.PLAIN, 14));
 setLayout(gridbag);

 c.fill = GridBagConstraints.BOTH;
 c.weightx = 1.0;
 makebutton("Button1", gridbag, c);
 makebutton("Button2", gridbag, c);
```

```
 makebutton("Button3", gridbag, c);

c.gridwidth = GridBagConstraints.REMAINDER; //end of row
makebutton("Button4", gridbag, c);

          // reset to the default
c.weightx = 0.0;
makebutton("Button5", gridbag, c);  //another row

c.gridwidth =  GridBagConstraints.RELATIVE; //next to last in
row
makebutton("Button6", gridbag, c);

c.gridwidth= GridBagConstraints.REMAINDER; // end of row
makebutton("Button7", gridbag, c);

c.gridwidth = 1;                  //reset to the default
c.gridwidth = 2;
c.weighty = 1.0;
makebutton ("Button8", gridbag, c);

c.weighty = 0.0;      //reset to default
c.gridwidth = GridBagConstraints.REMAINDER;//end of row
c.gridheight = 1;

makebutton("Button9", gridbag, c);
makebutton("Button10", gridbag, c);
}
protected void makebutton(String name, GridBagLayout gridbag,
GridBagConstraints c)
{
Button button = new Button(name);
gridbag.setConstraints(button, c);
add(button);
}
}
```



11.8.5 MOUSE EXAMPLE

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.awt.event.MouseEvent.*;

public class mouse2 extends Applet implements MouseListener
{

 TextField T1, T2;
 int x, y, x1, y1;
 Butto b1,b2,b3;
public void init()
{
  T1 =  new TextField(20);
  add(T1);
  T2 = new TextField(20);
 add(T2);
  addMouseListener(this);
}
public void mouseClicked(MouseEvent e)
{}
public void mousePressed(MouseEvent e )
{
 System.out.println("Mouse pressed event is fired");
 x = e.getX();
 y = e.getY();
 T1.setText("X : " +Integer.parseInt(String.valueOf(x)) + "
Y : " +Integer.parseInt(String.valueOf(y)));
}
```

```
 public void mouseReleased(MouseEvent e )
{
System.out.println("Mouse released event is fired");
 x1 = e.getX();
 y1=  e.getY();
  T2.setText("X1 : " +Integer.parseInt(String.valueOf(x1)) +
" Y1 : " +Integer.parseInt(String.valueOf(y1)));
repaint();
}
  public void paint(Graphics g)
{
 g.drawLine(x, y, x1, y1);
}

public void mouseEntered(MouseEvent e)
{
 T1.setText("Mouse is in");
}

public void mouseExited(MouseEvent e)
{
 T1.setText("Mouse is out");
}
}
```

11.8.6 MENU FRAME EXAMPLE

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

public class Dialog1 extends Applet {

     Button button1, button2;
     menuFrame frameWindow;
     public void init(){
          button1 = new Button("Show password window");
          add(button1);
          button2 = new Button("Hide password window");
          add(button2);
          frameWindow = new menuFrame("Password");
          frameWindow.resize(300,100);
```

```
        }

     public boolean action (Event e, Object o){
          if(e.target.equals(button1)){
               frameWindow.show();
               }
          if(e.target.equals(button2)){
               frameWindow.hide();
               }
          return true;
          }
}

class menuFrame extends Frame {
     TextField NameText, PasswordText;
     Label Namelabel, Passwordlabel;
     Dialog dialogBox;
     Button B1, B2;

menuFrame(String title){
     super(title);
     setLayout(new GridLayout(3, 2));
     Namelabel = new Label(" Enter Name");
     add(Namelabel);
     NameText = new TextField(20);
     add(NameText);
     Passwordlabel = new Label(" Enter Password");
     add(Passwordlabel);
     PasswordText = new TextField(20);
     add(PasswordText);
     B1 = new Button("OKButton");
     add(B1);
     B2 = new Button("CancelButton");
     add(B2);
     dialogBox = new OKDialog(this, "Dialog Box", true);
     dialogBox.resize(200, 200);
 }

public boolean action (Event e, Object o){
     String caption = (String)o;
     if(e.target instanceof Button){
          if(caption == "OKButton"){
```

```
                dialogBox.show();
                }
           if(caption == "CancelButton"){
                NameText.setText("        ");
                PasswordText.setText("      ");
                }
           }
            return true;
     }
}

class OKDialog extends Dialog {

Button OKB, CanB;
TextField T1;
OKDialog(Frame hostFrame, String title, boolean dModal){
     super(hostFrame, title, dModal);
     setLayout(new GridLayout(2, 1));
     T1 = new TextField(20);
     add(T1);
     T1.setText("Password accepted");
     OKB = new Button("OK");
     add(OKB);
     CanB = new Button("Cancel");
     add(CanB);
}
public boolean action (Event e, Object o){
     String caption = (String)o;
     if(e.target instanceof Button){
          if(caption == "OK"){
                hide();
                }
          if(caption == "Cancel"){
                hide();
                }
     return true;
     }
}
```

Chapter 12 - Threads and Multithreading

If the program execution is blocked while waiting for the completion of some I/O operation, no other portion of the program can proceed. However, the users of today's modern operating systems are accustomed to starting multiple programs and watching them working concurrently, even if there is only a single CPU available to run all the applications. Multithreading allows different tasks to execute concurrently within a single program.

The advantage of Multithreading is twofold. First, program with multiple threads will, in general, better utilize system resources, including the CPU, because another line of execution can grab the CPU when one line of execution is blocked. Second, multiple threads solve Numerous problems better. For example, how would you write a single – threaded program to show animation, play music, display documents, and download files from the network at the same time?

Java was designed from the beginning with multithreading in mind. Not only does the language itself have multithreading support built in, allowing for easy creation of robust, multithreaded applications, but the virtual machine relies on multithreading to concurrently provide multiple services – like garbage collection – to the application. In this chapter, you will learn to use multiple threads in your Java programs.

12.1 Overview of Multithreading –

A thread is a single flow of control within a program. It is sometimes called the execution context because each thread must have its own resources - like the program counter and the execution stack-as the context for execution. However, all the threads in a program still share many resources, such as memory space and opened files. Therefore, a thread may also be called a lightweight process. It is a single flow of control like a process (or a running program), but it is easier to create and destroy than a process because less resource management is involved.

A program may spend a big portion of its execution time just waiting. For example, it may wait for some resource to become accessible in an I/O operation, or it may wait for some time-out to occur to start drawing the next scene of an animation

sequence. To improve CPU utilization, all the tasks with potentiality long waits can run as separate threads. Once a task starts waiting for something to happen, the Java run time can choose another runnable task for execution.

12.2 THREAD BASICS

The following sections introduce the basics of working with threads, including how to create and run threads, control thread executions, and get information about threads and thread groups. You will also learn about the life cycle of a thread and thread groups.

12.2.1 Creating and Running a Thread

When you have a task you want to run concurrently with other tasks, there are two ways to do this create a new class as a subclass of the Tread class or declare a class implementing the Runnable interface.

Using a Subclass of the Thread Class

When you create a subclass of the Thread class, this subclass should define its own run () method to override the run () method of the Thread class. This run() method is where the task is performed.

Just as the main () method is the first user defined methods the Java run time calls to start a thread. An instance of this subclass is then created by a new statement, followed by a call to the thread's start () method to have the run () method executed.

Implementing the Runnable Interface

The Runnable interface requires only one method to be implemented the run() method. You first create an instance of this class with new() statement, followed by the creation of a Thread instance with another new statement, and finally a call to this thread instance's start () method to start performing the task defined in the run() method. A class instance with the run() method defined within it must be passed in as an argument in creating the Thread instance, so that when the start() method of this Thread instance is called, Java run time knows which run() method to execute.

This alternative way of creating a thread comes in handy when the class defining the run() method needs to be a subclass of another class. The class can inherit all the data and methods of the super class, and the Thread instance just created can be used for thread control.

12.2.2 The Thread –Control Methods

Many methods defined in the java.lang.Thread class control the running of a thread. Java 2 deprecated several of them to prevent data inconsistencies or deadlocks. If you are just starting Java 2, avoid the deprecated methods and use the equivalent behavior described later in this chapter. However, if you are transitioning from Java 1.0 or 1.1, you will need to modify your code to avoid the deprecated methods if you used them. Here are some of ones that were most commonly used.

void start() throw IllegalThreadStateException

Used to start the execution of the thread body defined in the run() method. Program control will be immediately returned to the caller, and a new thread will be scheduled to execute the run() method concurrently with the caller's thread.

Static void sleep(long slipTimeInMilliseconds) throws InterruptedException

A class method that causes the Java run time to put the caller thread to sleep for a minimum of the specified time period. The InterruptedException may be thrown while a thread is sleeping or anytime if you interrupt() it. Either a try-catch statement needs to be defined to handle this exception or the enclosing method needs to have this exception in the throws clause.

Void join() throws InterruptedException

Used for the caller's thread to wait for this thread to die-for example, by coming to the end of the run() method.

Static void yield()

A class method that temporarily stops the caller's thread and puts it at the end of the queue to wait for another turn to be executed. It is used to make sure other threads of the same priority have a chance to run.

12.3 THE THREAD LIFE CYCLE

Every thread, after creation and before destruction, will always be in one of four states: newly created, Runnable, blocked, or dead. These states are illustrated in the fig.

12.3.1 Newly Created Threads

A thread enters the newly created state immediately after creating; that is, it enters the state right after the thread –creating new statement is executed. In this state, the local data members are allocated and initialized, but execution of the run () method will not begin until its start () method is called. After the start () method is called, the thread will be put into the runnable state.

12.3.2 Runnable Threads

When a thread is in the runnable state, the execution context exists and the thread can be scheduled to run at any time; that is the thread is not waiting for any event to happen.

For the sake of explanation, this state can be subdivided into two sub states: the running and queued states. When a thread is in the running state, it is assigned CPU cycles and is actually running. When a thread is in the queued state, it is waiting for the queue and competing for its turn to spend CPU cycles. The virtual machine scheduler controls the transition between these two subclasses. However, a thread can call the yield() method to voluntarily move itself to the queued state from the running states.

12.3.3 Blocked Threads

The blocked state is entered when one of the following events occurs:

- - The thread itself or another thread calls the suspend () methods.
- - The thread calls an object’s wait () method.
- - The thread itself calls the sleep() method.
- - The thread is waiting for some I/O operation to complete.
- - The thread will join () with another thread.

A thread in a blocked state will not be scheduled for running. It will go back to the runnable state, competing for CPU cycles, when the counter event for the blocking event occurs:

- - If the thread is suspended, another thread calls its resume () methods.
- - If the thread is blocked by calling an object's wait () method, the object's notify () or notifyAll () method is called.
- - If the thread is put to sleep, the specified sleeping time elapses.
- - If the thread is blocked on I/O, the specified I/O operation completes.

12.3.4 Dead Threads

- The dead state is entered when a thread finishes its execution or is stopped by another thread calling its stop() method.
- To avoid the use of stop(), the proper way to exit out of a while (true) loop is to maintain a state variable that is used as the while loop condition check. So instead of run() looking like the following and using stop() to halt the thread:
- Public void run() {
- While (true) {
    - …
  - }
- }
-

You would change the test case to be some boolean condition. Then, when you want the thread to stop , instead of calling stop() , you change the state of the boolean. This causes the thread to stop on the next pass and ensures that the thread doesn't leave data in an inconsistent state.

```
Public void run() {
        While (aBooleanVariable) {
                ….
                }
        }
```

12.4 THREAD GROUPS

Every thread instances is a member of exactly one thread group. A thread group can have both threads and other thread groups a its members. In fact, every thread group, except the system thread group, is a member of some other thread group. All the threads and thread groups in an application form a tree, with the systems thread group as the root.

When a Java applications is started, the Java virtual machine creates the main thread group as a member of the systems thread group. A main thread is created in this main thread group to run main() method of the application. By default, all new user-created threads and thread groups will become the members of this main thread group unless another thread group is passed as the first argument of the constructor method of the new statement. A new thread group is created by instantiating the ThreadGroup class. For example, the following statements create a thread group named MyThreadGroup as a member of the default main thread group, and then create a thread named MyThread as a member of the newly created thread group.

```
ThreadGroup group = new ThreadGroup (“MyThreadGroup”);
Thread thread = new Thread (group, “MyThread”);
```

Three method are defined in the ThreadGroup class to manipulate all the thread in the Thread class group and its subthread groups at once: stop(), suspend(), and resume(). As with Thread, the stop(), suspend(), and resume() methods are now deprecated and should be avoided. (basically, the methods has a use that didn’t work properly, so Sun deprecated them.) Proper use of condition variable should diminish the dependency on these methods.

12.5 GETTING INFORMATION ABOUT THREADS AND THREAD GROUPS

Many method are defined in Thread and ThreadGroup for getting information about threads and thread groups.

12.5.1 Thread Information

The following are some of the most commonly used methods for getting information about threads:

java. lang. thread method:

static thread currentThread() Return the caller's thread.

String getName() Return the current name of the thread.

ThreadGroup getThreadGrouop() Returns the parent thread group of the thread.

int getPriority() Returns the current priority of the thread

boolean isAlive() Returns true if the thread is started but not dead yet.

boolean isDaemon() Returns true if the thread is a daemon thread.

12.5.2 Thread Group Information

The following are some of the most commonly used methods for getting information about thread groups:

java. lang. Threadgroup methods:

String getName() Returns the name of the thread group.

ThreadGroup getParent() Returns the parent thread Group of the thread group.

int getMaxPriority() Returns the current maximum priority of the thread group.

int activeCount() Returns the number of active threads in the thread group.

int activeGroupCount() Returns the number of active thread groups in the group.

int enumerate (Thread list [], boolean recursive)

Adds all the active threads in this group into the list array. If recursive is true, all the threads in the subthread groups will be copied over as well. This method will return the number of thread copied. The activeCount() method is often used to size the list when the space of this thread array is to be allocated.

12.6 THREAD SYNCHRONIZATION

Synchronization is the way to avoid data corruption caused by simultaneous access to the same data. Because all the threads in a program share the same memory space, it is possible for two threads to access the same variable or run the same method of the same object at the same time.

Problem may occur when multiple threads are accessing the same data concurrently. Threads may race each other, and one thread may overwrite the data just written by another thread. Or one thread may work on another thread's intermediate result and break the consistency of the data. Some mechanism is needed to block one thread's access to the critical, if the data is being worked on by another thread.

12.6.1 Java's Monitor Model for Synchronization

Java uses the idea of monitors to synchronize access to data. A monitor is like a guarded place where all the protected resources have the same locks. Only a single key fits all the locks insides a monitor, and a thread must get the key to enter the monitor and access theses protected resources. If many threads want to enter the monitor simultaneously, only one thread is handed the key; the others must wait outside until the key-holding thread finishes its use of the resources and hands the key back to the Java virtual machines.

Once a thread gets a monitor's key, the thread can access any of the resources controlled by that monitor countless times, as long as the thread still owns the key. However, if this key-holding thread wants to access the resources controlled by another monitor, the threads must get that particular monitor's key. At any time, a thread can hold many monitors' keys. Different threads can hold keys for different monitors at the same time. Deadlock may occur if threads are waiting for each other's key to proceed.

In Java, the resources protected by monitors are program fragments in the form of methods or blocks of statements enclosed in curly braces. If some data can be accessed only through method or blocks protected by the same monitor, access to the data is indirectly synchronized. You use the keyword synchronized to indicate that the following method or block of statements is to be synchronized, an object instance enclosed in parentheses immediately following the synchronized keyword is required so the Java virtual machine knows which monitor to check.

You can think of a monitor as a guarded parking lot , where all the synchronized methods or blocks are just like cars you can drive (or execute , if you are a thread).

All the cars share same key. You need to get this unique key to enter the parking lot and drive any of the cars until you hand back the key. At that time, one of the persons waiting to get in will get the key and be able to drive the car(s) of their choice.

synchronized void deposit(int amount, String name) {}

12.7 INTER-THREAD COMMUNICATIONS

Inter-thread communications allow threads to talk to or wait for each other. You can have threads communicate with each other through shared data or by using thread-control methods to have threads wait for each other.

12.7.1 Threads Sharing Data

All the threads in the same program share the same memory space. If the reference to an object is visible to different threads by the syntactic rules of scopes, or explicitly passed to different threads, these threads share the access to the data members of that object. Synchronization is sometimes necessary to enforce exclusive access to the data to avoid racing conditions and data corruption.

12.7.2 Threads Waiting for Other Threads

By using thread-control methods , you can have threads communicate by waiting for each other. For example, the join() method can be used for the caller thread to wait for the completion of the called thread. Also, a thread can suspend itself and wait at a rendezvous point using the suspend() method; another thread can wake it up through the waiting thread's resume() method, and both threads can run concurrently thereafter.

Deadlock may occur when a thread holding the key to a monitor is suspended or waiting for another thread's completion. If the other thread it is waiting for needs to get into same monitor, both threads will be waiting forever. This is why the suspend() and resume() methods are now deprecated and should not be used. The wait() , notify(), and notifyAll() methods defined in class Object of the Java.lang package can be used to solve this problem.

The wait() method will make the calling thread wait until either a time-out occurs or another thread calls the same object's notify() or notifyAll() method . The synopsis of the wait() methods is:

```
Wait()
```

Or

```
Wait(long timeoutPeriodInMilliseconds)
```

The former will wait until the thread is notified. The later will wait until either the specified time-out expires or the thread is notified , whichever comes first.

When the thread calls the wait() method, the key it is holding will be released for another waiting thread to enter the monitor. The notify() method will wake up only one waiting thread , if any. The notifyAll() methods will wake up all the threads that have been waiting in the monitor. After being notified, the thread will try to reenter the monitor by requesting the key again and may need to wait for another thread to release the key.

Note that these methods can be called only within a monitor, or synchronized block. The thread calling an object's notify or notifyAll() method needs to own the key to that object's monitor, otherwise, IllegalMonitorStateException, a type of RuntimeException will be thrown.

12.8 PRIORITIES AND SCHEDULING –

Priorities are the way to make sure important or time-critical threads are executed frequently or immediately. Scheduling is the means to make sure priorities and fairness are enforced.

If you have only one CPU, all of the runnable threads must take turns executing. Scheduling is the activity of determining the execution order of multiple threads.

12.8.1 Thread Priority Values

Every thread in Java is assigned a priority value. When more than one thread is competing for CPU time, the thread with the highest priority value is given preference. thread priority values that can be assigned to user-created threads are simple integers ranging between Thread.MIN_PRIORITY and Thread.MAX_PRIORITY. User applications are normally run with the priority value

of Thread.NORM_PRIORITY. Through JDK 1.2, the following constants of the Thread class -MIN_PRIORITY, MAX_PRIORITY , and NORM_PRIORITY- have the values of 1,10, and 5, respectively. Every thread group has a maximum priority value assigned. This is a cap to the priority values of member thread groups when they are created or want to change their priority values.

When a thread is created, it will inherit the priority value of the creating thread if the priority values doesn't exceed the limit imposed by its parent thread. The setPriority() method of Thread class can be used to set the priority value of a thread. If the value to be set is outside the legal range, an IllegalArgumentException will be thrown. If the value is larger than the maximum priority value of its parent thread group, the maximum priority value will be used.

The setMaxPriority() method of class ThreadGroup can be used to set the maximum priority value of a thread group. For security reasons (so that a user-created thread will not monopolize the CPU) , Web browser may not allow an applet to change its priority.

12.8.2 Preemptive Scheduling and Time-Slicing

Java's scheduling is preemptive; that is , if a thread with a higher priority than the currently running thread becomes runnable, the higher priority thread should be executed immediately, pushing the currently running thread back to the queue to wait for its next turn. A thread can voluntarily pass the CPU execution privilege to waiting threads of- the same priority by calling the yield() method.

In some implementations, thread execution is time-sliced; that is, threads with equal priority values will have equal opportunities to run in a round-robin manner. Even threads with lower priorities will still get a small portion of the execution time slots, roughly proportional to their priority values. Therefore, no threads will be starving in the long run.

Other implementations do not have time-slicing. A thread will relinquish its control only when it finishes its execution, is preempted by a higher-priority thread or is blocked by I/O operations or the sleep(), wait(), or suspend() method calls. For computation-intensive threads, it is a good idea to occasionally call the yield()

method to give other threads a chance to run. It may improve the overall interactive responsiveness of graphical user interfaces.

12.9 DAEMON THREADS

Daemon threads are service threads. They exist to provide services to other threads. They normally enter an endless loop waiting for clients requesting services. When all the active threads remaining are daemon threads, the Java virtual machine will exit.

For example, a timer thread that wakes up in regular intervals is a good candidate for daemon threads. This timer thread can notify other threads regularly about the time-outs. When no other thread is running there is no need for the timer thread's existence

To create a daemon thread, call the setDeamon() method immediately after the thread's creation and before the execution is started. The constructor of the thread is a good candidate for making this method call. By default, all the threads created by a daemon thread are also daemon threads. The synopsis of the setDaemon() method is:

```
setDaemon (boolean isDaemon)
```

When isDaemon is true, the thread is marked as a daemon thread; otherwise , it is marked as a non-daemon thread.

Chapter 13 - Java DataBase Connectivity(JDBC)

Java offers several benefits to the developer creating a front-end application for a database server. Java is a “write once, run anywhere” language. This means that Java programs may be deployed without recompilation on any of the architecture and operating systems that possesses a Java Virtual Machine. For large corporations, just having a common development platform is a big saving: no longer are programmers required to write separate applications for the many platforms a large corporation may have. Java is also attractive to third party developers – a single Java program can answer the needs of both large and small customers.

This chapter examines Java database Connectivity (JDBC), including how to use the current JDBC API to connect Java applications and applets to database servers.

13.1 THE JDBC API

The JDBC API is designed to allow developers to create database front ends without having to continually rewrite their code. Despite standards set by the ANSI committee, each database system vendor has a unique way of connecting and, in some cases, communicating with their system.

The ability to create robust, platform-independent applications and Web-based applets prompted developers to consider using Java to develop front-end connectivity solutions. At the outset, third party software developers met the need by providing proprietary solutions, using native methods to integrate client-side libraries or creating a third tier and a new protocol.

The Java Software Division, Sun Microsystems’ division responsible for the development of Java products, worked in conjunction with the database and database-tool vendors to create a DBMS-independent mechanism that would allow developers to write their client side applications without concern for the particular database being used. The result is the JDBC API, which is part of the core JDK1.2.

JDBC provides application developers with a single API that is uniform and database independent. The API provides a standard to write to, and a standard that considers all of the various application designs. The secret is a set of Java interfaces that are implemented by a driver. The driver takes care of the translation of the standard JDBC calls into the specific calls required by the database it supports. In the following fig., the application is written once and moved to various drivers. The application remains same; the drivers change. Drivers may be used to develop the middle tier of a multi-tier database design, also known as middleware.

JDBC is not a derivative Microsoft's Open Database Connectivity specification (ODBC). JDBC is written entirely in Java and ODBC is a C interface. While ODBC is usable by non-C languages, like Visual Basic, it has the inherent development risks of C, such as memory leaks. However, both JDBC and ODBC are based on the X/Open SQL Command Level Interface (CLI). Having the same conceptual base allowed work on the API to proceed quickly and make acceptance and learning of the API easier .Sun provides a JDBC-ODBC bridge that translates JDBC to ODBC. This implementation, done with native methods, is very small and efficient.

In general, there are two levels of interfaces in the JDBC API; the Application Layer, where the developer uses the API to make calls to the database via SQL and retrieve the results, and the Driver Layer, which handles all communication with a specific Driver implementation.

13.2 THE API COMPONENTS

The Application Layer, which database-application developers use, and the Driver Layer, which the driver vendors implement .It is important to understand the Driver Layer, if only to realize that the driver creates some of the objects used at the Application Layer. The Figure below illustrates the connection between the Driver and Application layers.

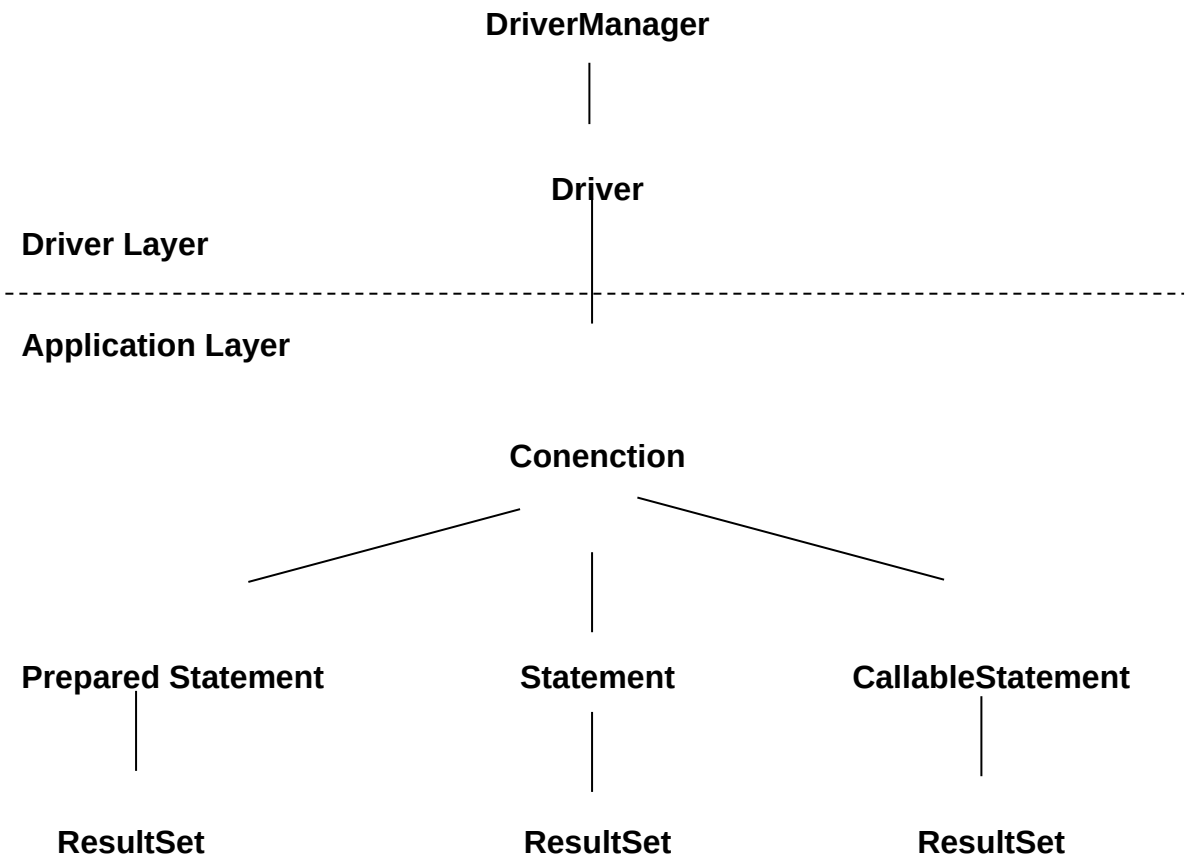


Figure: JDBC API Components

Fortunately the application developer need only use the standard API interface in order to guarantee JDBC compliance. The driver developer is responsible for developing code that interfaces to the database and supports the JDBC application level calls.

There are four main interfaces that every driver layer must implement, and one class that bridges the Application and Driver layers. The four interfaces are the Driver, Connection, Statement, and ResultSet. The driver interface implementation is where the connection to the database is made. In most applications, the Driver is accessed through the DriverManager class- providing one more layer of abstraction for the developer.

The Connection, Statement, and ResultSet interfaces are implemented by the driver vendor, but these interfaces represent the methods that the application

developer will treat as real object classes and allow the developer to create statements and retrieve results. So the distinction in this section between Driver and Application layers is artificial –but it allows the developer to create the database applications without having to think about where the object are coming from or worry about what specific driver the application will use.

13.2.1 The Driver Layer

There is a one-one correspondence between the database and the JDBC driver. This approach is common in multi-tier designs. The Driver class is an interface implemented by the driver vendor. The other important class is the DriverManager class, which sits above the Driver and Application layers. The DriverManager is responsible for loading and unloading drivers and making connections through drivers. The DriverManager also provides features for logging and database login timeouts.

The Driver Interface

Every JDBC program must have at least one JDBC driver implementation. The Driver interface allows the DriverManager and JDBC Application layers to exist independently of the particular database used. A JDBC driver is an implementation of the Driver interface class. Drivers use a string to locate and access databases. The syntax of this string is very similar to a URL string. The purpose of JDBC URL string is to separate the application developer from the driver developer.

The driver vendor implements the Driver interface by creating methods for each of the following interface methods:

Signature: public interface java.sql.Driver

`Public abstract Connection connect (String url, Properties info) throws SQLException`

The driver implementation of this method should check the subprotocol name of the URL string passed for a match with this driver. If there is a match, the driver should then attempt to connect to the database using the information passed in the remainder of the URL. A successful

database connection will return an instance of the driver's implementation of a Connection interface (object). The SQLException should be thrown only if the driver recognizes the URL subprotocol but cannot make the database connection. A null is returned if the URL does not match a URL, the driver expected. The username and password are included in a container class called Properties.

Public abstract Driver PropertyInfo [] getPropertyInfo (String url, Properties info) throws SQLException

If you are not aware of which properties to use when calling connect (), you can ask the Driver if the supplied properties are sufficient to establish a connection. If they are not an array of necessary properties is provided with the help of the DriverPropertyInfo class.

Public abstract Boolean acceptURL (string url) throws SQLException

It is also possible to explicitly "ask" the driver if a URL is valid.but note that the implementation of this method (typically) only checks if the subprotocol specified in the URL is valid, not wheather a connection can be made.

Public int getMajorVersion ()

Returns the drivers major version number. If the driver's version was at 4.3, this would return integer 4.

Public int getMinorVersion ()

Returns the drivers minor version number. If the driver's version was at 4.3, this would return integer 3.

Public Boolean jdbcComplaint ()

Returns whether or not the driver is a complete JDBC implementation. For legacy systems or lightweight solutions, it may not be possible, or necessary, to have to complete implementation.

The connect () method of driver is the most important method and is called by the DriverManager to obtain the connection object.as figure 15.5 previously

showed, the connection object is the starting point of JDBC application layer. The Connection object is used to create Statement objects that perform queries.

The connect () method typically performs the following steps.

1. Checks to see if the URL string provided is valid.
2. Opens a tcp connection to the host and port number specified
3. Attempts to access the named database table (if any)
4. Returns an instance of a connection object.

The DriverManager class

The DriverManager class is really a utility class used to manage JDBC drivers. The class provides method to obtain a connection through a driver, register and de-register drivers, set up logging, and set login timeouts for database access. All of the methods in the DriverManager class listed below are static, may be, and may be referenced through the following class name.

Signature: `public class java.sql.DriverManager`

`Public static synchronized connection getConnection (String url, Properties info) throws SQLException`

This method (and the other getconnection () methods) attempt to return a reference to an object implementing the connection interface. The method sweeps through an internal collection of stored driver classes, passing the URL string and properties object info to each in turn. The first driver class that returns a connection is used. Info is a reference to a properties container object of tag/value pairs, typically username/password. This method allows several attempts to make an authorized connection for each driver in the collection.

`Public static synchronized connection getConnection (String url) throws SQLException`

This method calls getConnection (url, info) above with an empty properties object (info).

Public static synchronized connection getConnection (string uql, string user, string password) throws SQLException

This method creates a properties object (info), stores the user and password strings in it, and then calls getConnection (url, info) above.

Public static synchronized void registerDriver (java.sql.Driver driver) throws SQLException

This method stores the instance of the driver interface implementation into a collection of drivers, along with the programs current security context to identify where the driver came from.

Public static void setLogStream (PrintStream out) deprecated.

This method sets a logging/tracing PrintStream that is used by DriverManager.

Public static void setLoginTimeout (int seconds)

This method sets the permissible delay a driver should wait when attempting a database login. Drivers are registered with the DriverManager class either at initialization of the DriverManager class or when an instance of the driver is created.

When the DriverManager class is loaded, a section of static code (in the class) is run, and the class names of drivers listed in a java property named JDBC.drivers are loaded. This property can be used to define a list of colon seperated driver class names, such as:

jdbc.drivers=imaginary.sql.Driver:oracle.sql.Driver: weblogic.sql.Driver

Each driver name is a class file name (including the package declaration) that the DriverManager will attempt to load through the current CLASSPATH. The DriverManager uses the following call to locate, load and link the named class:

```
Class.forName(driver);
```

If the jdbc.drivers property is empty (unspecified), then the application programmer must create an instance of driver class.

In both cases, the driver class implementation must explicitly register itself with the DriverManager by calling.

```
DriverManager.registerDriver (this);
```

13.2.2 The Application Layer

The application layer encompasses three interfaced that are implemented at the driver layer but are used by the application developer. In java, the interface provides a means of using a general name to indicate a specific object. The general name defines methods that must be implemented by the specific object. For the application developer, this means that the specific driver class implementation is irrelevant. Just coding to the standard JDBC, API's will be sufficient. This is of course, if the driver is JDBC complaint. Recall that this means the database at least supports ANSI SQL-2 entry level.

The three main interfaces are connection, statement and ResultSet. A connection method is obtained from the driver implementation through the drivermanager.getConnection () method call. Once a collection object is returned, the application developer may create a statement object to issue against the database.the result of a statement object to issue against the database. The result of a statement is a ResultSet object, which contains the result of the particular statement (if any).

Connection basics

The Connection interface represents a session with the database connection provided by the driver. Typical database connections include the ability to control changes made to actual data stored through transactions. A transaction is a set of operations that are completed in order. A commit action makes the operations store (to change) data in the database. A rollback action undoes the previous transaction before it has been committed. On creation, JDBC connections are in an auto-commit mode; there is no rollback possible. Therefore, after getting a connection object from the driver, the developer should consider setting auto-commit to false with setAutoCommit (Boolean b) method.

When auto-commit is disabled, the connection will support both connection.Commit () and connection.RollBack () method calls. The level of support for transaction isolation depends on the underlaying support for transaction in the support.

A portion of the connection interface definition follows:

Signature: `public interface connection`

`Statement createStatement() throws SQLException`

The connection object implementation will return an instance of an implementation of a statement object. The statement object is then used to issue queries.

`PraperedStatement prapareStatement(String sql) throws SQLException`

The connection object implementation will return an instance of a PraperedStatement object that. The driver may then send the statement to the database; if the database (driver) handles precompiled statements. Otherwise the driver may wait until the PraparedStatement is executed by an execute method. An exception may be thrown if the driver and database do not implement precompiled statements.

CallableStatement prapareCall (String url) throws SQLException

The connection object implementation will return an instance of a CallableStatement. CallableStatements are optimized for handling stored procedures. The driver implementation may send the SQL string immediately when prapareCall () is complete or may wait until an execute () method occures.

`Void setAutoCommit (Boolean autocommit) throws SQLException`

Sets a flag in the driver implementation that enables commit/rollback (false) or makes all transactions commit immediately (true).

`Void commit throws SQLException`

Makes all changes made since the beginning of the current transaction (either the opening of the connection or since the last commit () or rollback ())

Void rollback () throws SQLException

Drops all changes made since the beginning of the current transaction.

The primary use of the connection interface is to create a statement

```
Connection msqlConn;
Statement stmt;

MsqlConn=drivermanager.getConnection (url);
Stmt=msqlConn.creatStatement();
```

This statement may be used to send SQL queries that return a single result set in a ResultSet object reference or a count of the number of records affected by the statement. Statements that need to be called a number of times with slight variations may be executed more effectively using a PraparedStatement. The connection interface is also used to create a CallableStatement whose primary purpose is to execute stored procedures.

TIP

The primary difference between Statement, PreparedStatement, and Callable Statement is that Statement does not permit any parameters within the SQL statement to be executed, PreparedStatement permits In parameters, and CallableStatement permits Inout and Out parameters .In parameters are parameters that are passed into an operation .Due parameters are parameters passed by reference; they are expected to return a result of the reference type. Inout parameters are Out parameters that contain an initial value that may change as a result of the operation. JDBC supports all three parameter types.

Statement Basics

A statement is the vehicle for sending SQL queries to the database and retrieving a set of results. Statements can be SQL updates, inserts, deletes, or queries (via Select). The Statement interface provides a number of methods

designed to make the job of writing queries to the database easier. There are other methods to perform other operations with a Statement.

Signature: `public interface Statement`

`ResultSet executeQuery (String sql) throws SQLException`

Executes a single SQL query and returns the results in an object of type ResultSet.

`int executeUpdate (String sql) throws SQLException`

Executes a single SQL query that does not return a set of results, but a count of rows affected.

`boolean execute (Strinig sql) throws SQLException`

General SQL statements that may return multiple result sets and /or update counts. This method is most frequently used when you do not know what can be returned, probably because of a user entering the SQL statement directly. The getResultSet (), getUpdateCount (), and getMoreResults () methods are used to retreive the data returned.

`ResultSet getResultset() throws SQLException`

Returns the current data as the result of a statement execution as a ResultSet object. Note that if there are no results to be read or if the result is an update count, this method returns null. Also, note that the results are cleared.

`Int getUpdateCount () throws SQLException`

Returns the status of an Update, Insert, or Delete query or a stored procedure. The value returned is the number of rows affected. A –1 is returned if there is no update count or if the data returned is a result set. Once read, the update count is cleared.

`Boolean getMoreResults () throws SQLException`

Moves to the next result in a set of multiple results/update counts. This method returns true if the next result is a ResultSet object. This method will also close any previous ResultSet read.

Statements may or may not return a ResultSet object, depending on the Statement method used. The executeUpdate () method, for example, is used to execute SQL statements that do not expect a result (except a row count status):

```
int rowCount;
rowCount = stmt.executeUpdate (
"DELETE FROM customer WHERE CustomerID = 'McG10233'");
```

SQL statements that return a single set of results can use the executeQuery() method. This method returns a single ResultSet object. The object represents the row information returned as a result of the query.

```
Resultset results;
Results = stmt.executeQuery ("SELECT * FROM stock");
```

SQL statement that execute stored procedures (or trigger a stored procedure) may return more than one set of results. The execute() method is used a general-purpose method that can return a single result set, a result count, or some combination thereof. The method returns a Boolean flag that is used to determine where there are more results. Because a result set could contain either data or the count of an operation that returns a row count, the getResultSet(), getMoreResults(), and getUpdateCount() methods are used.

The PreparedStatement interface extends interface extends the Statement interface. When there is a SQL, statement that requires repetition with minor variations, the PreparedStatement provides an efficient mechanism for passing a precompiled SQL statement that uses input parameters.

Signature: `public interface PreparedStatement extends Statement`

PreparedStatement parameters are used to pass data into a SQL statement, so they are considered IN parameters and are filled in by using setType methods.

The CallableStatement interface is used to execute SQL stored procedures. CallableStatement inherits from the PreparedStatement interface, so all of the execute and setType methods are available. Stored procedures have varying

syntax among database vendors, so JDBC defines a standard way for all RDBMS to call stored procedures.

Signature: `public interface CallableStatement extends PreparedStatement`

The JDBC uses an escape syntax that allows parameters to be passed as In parameters and Out parameters. The syntax also allows a result to be returned; and if this syntax is used, the parameter must be registered as an Out parameter.

setType Methods

Method Signature	**Java Type**	**SQL Type From the database**
void setByte (int index, byte b)	byte	TINYINT
void setShort (int index, short s)	short	SMALLINT
void setInt (int index, int i)	int	INTEGER
void setLong (int index, long l)	long	BIGINT

getType methods

Method Signature	**Java Type**	**SQL Type From the database**
boolean getBoolean (int index)	boolean	BIT
byte getByte (int index)	byte	TINYINT
short getShort (int index)	short	SMALLINT
int getInt (int index)	int	INTEGER

ResultSet Basics

The ResultSet interface defines methods for accessing tables of the data generated as the result of executing a Statement. ReaultSet column values may be accessed in any order; they are indexed and may be selected by either the name or the number (numbered from one to n) of the first row of data returned. The next () method moves to the next row of data.

A partial look at the ResultSet interface follows:

Signature: `public interface ResultSet`

`boolean next () throws SQLException`

Positions the ResultSet to the next row; ResultSet row position is initially the first row of the result set.

`ResultSetMetaData getMetaData () throws SQLException`

Returns an object of the current result set; the number of columns, the type of each column, and properties of the results.

`void close () throws SQLException`

Normally a ResultSet is closed when another Statement is executed, but it may be desirable to release the resources earlier.

The 1.2JDK intorduces several methods with the JDBC2.0 API. With JDBC 2.0, additional capabilities are available that permit non-sequential reading of rows, as well as updating of rows, while reading. A partial look at the JDBC 2.0 methods of ResultSet follows:

`int getType () throws SQLException`

Returns the type of Result set, determining the manner in which you can read the results. Valid return values are TYPE_FORWARD-ONLY, TYPE-STATIC, TYPE_KEYSET, or TYPE_DYNAMIC .If TYPE_FORWARD_ONLY is returned most of the remaining methods will throw a SQLException if they are attempted.

`boolean first () throws SQLException`

Positions the ResultSet at the first row. Return true if on a valid row, false otherwise.

`boolean last () throws SQLException`

Positions the ResultSet at the last row. Return true if on a valid row, false otherwise

`boolean previous () throws SQLException`

Positions the ResultSet at the previous row.Return true if on a valid row, false otherwise

`boolean absolute (int row) throws SQLException`

Positions the ResultSet at the designated row. If the row requested is negative positions ResultSet at row relative to end of set.

boolean relative (Int row) throws SQLException

Positions the ResultSet at the row, relative to the current position.

boolean isFirst () throws SQLException

JDBC 2.0 indicates whether the cursor is on the first row of the ResultSet.

boolean isLast () throws SQLException

JDBC 2.0 indicates whether the cursor is on the last row of the ResultSet.

ResultSetMetaData besides being able to read data from a ResultSet object, JDBC provides an interface to allow the developer to determine what type of data was returned. The ResultSetMetaData interface is similar to the DatabaseMetaData interface in the concept, but is specific to the current ResultSet. As with DatabaseMetaData, it is unlikely that many developers will use this interface, since most applications are written with an understanding of the database schema and column names and values. However, ResultSetMetaData is useful in dynamically determining the meta-data of a ResultSet returned from a stored procedure or from a user-supplied SQL statement.

The following code demonstrates the displaying of results with the help of ResultSetMetaData when the contents are unknown:

```
ResultSet results = stmt.executeQuery (sqlString)
ResultsetMetaData meta = results.getMetaData ();
Int columns = 0;
boolean first = true;
while (results.next ()) {
     If (first) {
          columns = meta.getColumnCount();
          for (int i=1; i<=columns; i++)  {
```

```
        System.out.print     (metadata.getColumnName(i)   +
"\t");
        }
        System.out.println();
        first=false;
        }
        for (Int i=; i<=columns; i++)  {
        System.out.print (results.getString (i)+ "\t");
        }
        System.out.println();
        }
```