Chapter : 1

Introduction

1.0 OBJECTIVES

Dear Students,

After studying this chapter you will able to :

- discuss the background of development of Java
- explain with the Java buzzwords summed up by the Java committee.
- discuss what is meant by Object Oriented Programming.
- state the principles of Object Oriented Programming.

1.1 INTRODUCTION

Java was conceived by James Gosling, Patrick Noughton, Christ Warth, Ed Frank and Mike Sheridan at Sun Microsystems Inc. in 1991. The language was initially called **Oak** but later renamed as **Java** in 1995.

The primary need for the development of Java was the necessity of a platform independent language which could be used to create software for embedded systems. We know that languages like C and C++ can run on a variety of CPUs, however a compiler needs to be developed for every CPU on which the program is to be run. Compilers are expensive and time consuming to create. Therefore, during the development of Java the main emphasis was on creating a platform independent language which could run on a variety of CPUs under various environments. This led to the development of Java.

At around the same time the World Wide Web emerged and Java became a language of the Internet. The Internet consists of various types of computers operating under different operating systems and CPUs. It is necessary to run the same program on all these environments and Java had the capability to run on various platforms and thus offer portable programs. The Internet ultimately led to the phenomenal success of Java. Java has an immense effect on the Internet. In a network, two broad categories of objects are transmitted between the server and the client. The first category is the passive information and the second one is dynamic, self executing programs. Such a dynamic program is generated by the server, however it is active on the client computer. As such, these network programs pose serious difficulties in terms of security and portability.

We can create two types of programs in Java, **applications** and **applets**. An **application** is a program which runs on your computer under the operating system of your computer . An **applet** is an application which is developed to be transmitted over the Internet and executed by a Java compatible web browser. An **applet** is an intelligent program, which means it can react to the user input.

We know that everytime we try to download programs, we risk the possibility of infecting our systems with virus. Also other programs may exist which may try to gather private information such as credit card numbers, bank account numbers, passwords etc. by searching our computer files. Java provides a security against both these attacks by providing a **firewall** between the network application and the user's personal computer. This ability to maintain the security of the user's computer is considered to be one of the most important aspects of Java.

As we have already seen Java also satisfies the need of generating portable executable code. Both these aspects of security and portability are addressed by Java because the output generated by Java compiler is not executable code, but **bytecode**.

Bytecode is a set of instructions which are designed to be executed by the Java run time system called the **Java Virtual Machine**. Thus the **JVM** is the interpreter for the bytecode. Only the JVM needs to be implemented for each platform on which the Java programs are to be run. Although the details of JVM differ from machine to machine all of them interpret the same bytecode. This helps in creating portable programs which can run on a variety of CPUs. The execution of a Java program is under the control of the JVM, therefore it makes the program secure and ensures that no side effects outside the system are generated. Safety is also enhanced by the features provided in the language itself. The **bytecode** also makes the Java run time system execute programs faster.

1.1 Check Your Progress.

1. Fill in the blanks.

a) __________ is a set of instructions designed to be executed by the Java run time system.

b) The ___________ is the interpreter for bytecode.

c) Java is a ____________ independant language.

d) An applet is a dynamic program which is active on the _______ computer.

1.2 JAVA BUZZWORDS

This section describes briefly the list of buzzwords as summed up by the Java team. From among the buzzwords two viz. security and portability have already been addressed in the previous section. Here we shall see the remaining ones.

Simple : Java was designed to be easy to learn and effective to use. With prior programming experience, it is not difficult to master Java. Java provides a small number of clearly defined ways to accomplish a given task. Java inherits the syntax of C and C++ and many of the object oriented features of C++. Therefore users who have already learnt these languages find it easy to master Java with very little effort.

Object Oriented : Java is not designed to be source code compatible with any other language. The object model in Java is simple and easy to extend. The simple types, like integers are kept as high performance non objects.

Robust : The ability to create robust programs was given a very high priority during the design of Java. Java is a strictly typed language and restricts the programmer in a few key areas. It checks the program code at compile time as well as at run time. Java manages memory allocation and deallocation on its own unlike in C/C++ where the programmer has to manually allocate and free all dynamic memory. In Java, deallocation is completely automatic. Java provides object oriented exception handling.

Multithreaded : Java was designed to meet the real world requirement of creating interactive networked programs. Therefore to accomplish this, Java supports multithreading. Multithreaded programming allows you to write programs that can do many things simultaneously.

Architecture Neutral : Upgradation of operating systems, processor upgrades, and changes in the core system resources can all be the factors which may cause a program which was running successfully to malfunction at a later point of time. Java and the JVM has been designed in such a way so as to attempt to overcome this situation, which would enable the programs to run inspite of severe environmental changes.

Interpreted : We know now that the output of the Java compiler is the **bytecode** which is platform independent. This code can be interpreted by any system which provides a **Java Virtual Machine**.

Distributed : Java handles TCP/IP protocols. Accessing a resource using URL is similar to accessing a file. Java has a package called **Remote Method Invocation** (**RMI**) which brings a very high level of abstraction to client/server programming.

Dynamic : Java programs have the capability to carry a sufficient amount of run time type information. This information can be used to verify and resolve accesses to the objects at run time. This makes it possible to dynamically link code in a safe manner.

1.2 Check Your Progress.

1. Write True or False.

a) Java is a strictly typed language.

b) Multithreaded programming allows to write programs that do many things simultaneously.

c) Java does not check program code at run time.

1.3 OBJECT ORIENTED PRORAMMING

Object oriented programming is at the core of Java. All Java programs are object oriented. Let us therefore study the Object Oriented Programming Principles before commencing writing programs in Java.

We have already studied that all computer programs consists of two elements : code and data. A program can be organised either around its code or around its data. When a program is organised around the code we call it as a **process oriented** model where code acts on data. The process oriented model has been successfully implemented in procedural languages like C. However, as the programs grow in size they become more complex. In the second approach which is called **object oriented** programming, a program is organised around its data i.e **objects** and a set of interfaces to that data. Thus an object oriented program is characterized as data controlling the access to the code.

An important element of object oriented programming is **abstraction**. Abstraction allows you to use a complex system without being overwhelmed by its complexity. A powerful way of managing abstraction is through the use of hierarchical classification. The data from a traditional process oriented program is transformed into its component objects through abstraction. A sequence of process steps then becomes a collection of messages between these objects. Each object defines its own unique behaviour. These objects are treated as concrete entities. Your tell them to do something by sending messages. This forms the basis of object oriented programming.

1.3.1 OOP Principles :

Object oriented programming languages provide a mechanism which helps to implement the object oriented model. These are given herewith :

Encapsulation : This is the mechanism which binds the code and the data that it manipulates. It thus keeps both the code and the data safe from outside interference and misuse. Access to the code and data is possible only through well defined interfaces. With encapsulation everyone knows how to use the code regardless of the details of the implementation. The basis of encapsulation in Java is the **class**. The class defines the structure and behaviour i.e. data and code that will be shared by a set of objects. Each object of a given class contains the structure and behaviour defined by the class. The objects are referred to as an **instance** of a class. Thus a class is a logical construct and an object is a physical reality.

The code and the data that constitute a class are collectively called as **members** of the class. The data defined by the class are referred to as **member variables** or **instance variables**. The code which operates on the data are referred to as **member methods** or **methods**. The methods define how the member variables are used.

Inheritance : This is the process by which an object acquires the properties of another object. Inheritance supports the concept of **hierarchical classification**. With the use of inheritance, an object would need to define only those qualities that make it unique within its class. It can inherit the general attributes from its parent class. eg. if you wanted to describe animals in an abstract manner you would say they have the attributes like size, intelligence etc. They also have certain behavioural pattern like they eat, breathe, sleep. Such attributes and behaviour makes up the class definition of animals. A more specific class of animals could be mammals with some specific attributes like mammary glands. This class would be a subclass of animals and the animals class is the superclass. Mammals would inherit all the properties of the animal class and have its own specific attributes also. You can have more than one level of inheritance where a deeply inherited subclass will inherit all the attributes from each of its ancestors in the class hierarchy. If a given class encapsulates some attributes then any subclass will have the same attributes i.e it will inherit all the attributes of all its ancestors, as well as add its own specific characteristics.

Polymorphism : This is a feature which allows one interface to be used for a general class of actions. Polymorphism is expressed as “one interface, multiple methods”. Thus it is possible to design a generic interface to a group of related activities. Polymorphism helps in reducing complexity by allowing the same interface to be used to specify a general class of action. The compiler would select the specific method as it applies to each situation. The programmer needs to know and use only the general interface.

Thus through the application of object oriented principles, the various parts of a complex program can be brought together to form a cohesive, robust and maintainable whole. Every Java program involves encapsulation, inheritance and polymorphism. This means that the student should bear in mind that every Java program is object oriented and study how this is applied in the chapters that follow.

1.3 Check Your Progress.

1. Match the following.

	Column A	**Column B**
a)	Polymorhphism	(i) is a logical construct
b)	class	(ii) One interface, multiple methods
c)	Inheritance	(iii) is a physical reality
d)	object	(iv) supports the concept of hierarchical classification

1.4 SUMMARY

Java was conceived by James Gosling, Patrick Noughton, Christ Warth, Ed Frank and Mike Sheridan at Sun Microsystems Inc. in 1991. The language was initially called Oak but later renamed as Java in 1995.

Java provides security by a firewall between the network application and the user's personal computer. This ability to maintain the security of the user's computer is considered to be one of the most important aspects of Java.

The output generated by Java compiler is not executable code, but bytecode. Bytecode is a set of instructions which are designed to be executed by the Java run time system called the Java Virtual Machine. Thus the JVM is the interpreter for the bytecode. Although the details of JVM differ from machine to machine all of them interpret the same bytecode.

Following is the list of buzzwords as summed up by the Java team.

Simple, Object Oriented, Robust, Multithreaded, Architecture Neutral, Interpreted, Distributed, Dynamic, Object oriented, OOP Principles, Encapsulation, Inheritance, Polymorphism

Source : *blog.ibeesolutions.com*

1.5 CHECK YOUR PROGRESS - ANSWERS

1.1 1. a) Bytecode

b) Java Virtual Machine

c) platform

d) client

1.2 1. a) True

b) True

c) False

1.3 1. a) - (ii)

b) - (i)

c) - (iv)

d) - (iii)

1.6 QUESTIONS FOR SELF - STUDY

1. Describe the evolution of the Java Programming language.
2. Write a short note on the Java Buzzwords
3. What is byte code? What do you understand by JVM?
4. Explain Object oriented programming. Describe in brief the OOP principles.

1.7 SUGGESTED READINGS

1. www.java.com
2. www.freejavaguide.com
3. www.java-made-easy.com
4. www.tutorialspoint.com
5. www.roseindia.net

❑ ❑ ❑

Notes

Chapter : 2

Beginning With Java

2.0 OBJECTIVES

Dear Students,

After studying this chapter you will able to :

- Discuss the keywords, identifiers and literals of the Java language
- Explain how to write simple programs in Java and compile them
- Discribe the various data types in Java
- Discuss variables and scope rules of variables
- Explain understand how arrays are implemented in Java
- State the different operators in Java and operator precedences

2.1 INTRODUCTION

We begin our study of Java by understanding the basics like **keywords**, **identifiers** and **literals**. We also study the simple data types viz. **integers**, **floats** and **characters** and the different types of operators supported by Java in this chapter.

Let us begin our study of Java by writing a small program to understand how to compile and run a Java program.

```
/* A first sample program
this is the way to give multiline comments */
class First {
        //Another way of writing comments in the program
public static void main(String args[])
{
        System.out.println("My first program in Java");
}
}
```

In Java, the source file is called a **compilation unit**. This is a text file which will contain one or more **class** definitions. The extension to the filename should be **.java**. Note that this extension is four characters long. All the program code should reside in a class. By convention the name of the class should match the name of the file that holds the program. This means that the program illustrated above should be stored with a file name **First. java**. Also remember that Java is case sensitive. Therefore care has to be exercised while naming the file. To compile the above program execute the Java compiler **javac** at the command line as shown below:

```
C:> javac First.java
```

The Java compiler creates a file called **First.class**. This file contains the **bytecode** form of the program. This **bytecode** is interpreted by the interpreter. This means that the output of the java compiler is not a code which can be directly executed. Hence to actually run the program, you have to use the Java interpreter called **java**. You can do this by executing the command as shown below :

```
C:> java First
```

When the program runs you will get the following output :

```
My first program in Java
```

When you execute the Java interpreter as shown above, you are actually supplying the name of the class that you want the interpreter to execute. The interpreter automatically searches for the file by the specified name and which has a **.class** extension.

Now that you have seen how to successfully compile the program, let us begin our study of the Java programming language. The student must note here that those sections where Java has similarities with languages like C and C++ shall be explained in brief and only an overview may be presented. Detailed programs may not be presented for every subtopic. Also wherever Java differs from these languages will be specifically indicated throughout the course material.

Let us study the line :

class First

We use the keyword **class** to declare that we are defining a new class. **First** is the name we have given to the class. Thus First is an **identifier**. The entire class definition, which includes all the members of that class is to be enclosed in a pair of curly braces. This is identical to the way they are used in C and C++.

The next line of code is :

public static void main(String args[])

This line begins the **main()** method. This is the line where program execution begins. At this point of time we shall not attempt to study the details of this line. It is enough to note that your program execution begins at this line and that all Java applications begin execution by calling **main()**. As we progress through our study we shall study this line in detail. It is also important to note that the Java compiler will compile classes that do not contain the **main()** method, however the Java interpreter cannot run these classes.

The brace bracket indicates the start of the body of the **main()** method. All the code of a particular method should be enclosed within a pair of braces.

The line -

System.out.println(“My first program in Java”);

will output

My first program in Java

on the screen. We use the **println()** method to output information on the console. In our example, **println()** outputs a string on the screen. **System** is a predefined class which provides access to the system and **out** is the output stream connected to the console. You should note that console input/output is mostly used for simple, utility programs. Modern computing environments are windowed and graphical in nature, therefore console input/output is not frequently used in real Java programs and applets. Also note that the **println()** statement ends with a semicolon (;). All statements in Java must end with a semicolon.

A Java program is a collection of comments, whitespace, identifiers, keywords, literals, separators, and operators. Now that we have studied a simple Java program, let us begin our study of these elements.

2.1 Check Your Progress.

1. Fill in the blanks.

a) The extension to a Java source file should be _________ .

b) The Java compiler creates a file which has the ________ extension, after compiling the source file.

2. Write True or False

a) All program code in Java should reside in a class.

b) A Java compiler compiles classes which do not contain a **main()** method.

c) Output of the Java compiler is a code which can be directly executed.

2.2 COMMENTS, WHITESPACES, SEPARATORS

We have already studied that anything that has been entered as a comment is ignored by the compiler. Comments are useful to explain the operation of the program to anyone who is reading it. Java supports three types of comments.

2.2.1 Comments

Multiline comments : These begin with a **/*** and end with ***/**. Anything that is written between these two comments symbols is a comment and is ignored by the compiler. As the name suggests, these comments can be several lines long.

Single line comments : These begin with the symbol **//** and end at the end of line. Such types of comments do not have an ending symbol.

Documentation comments : These begin with /** and end with */. This type of comment is used to produce and HTML file that documents the program. We shall not attempt to study this in detail.

2.2.2 White spaces

Java is a free form language which means that there are no specific indentation rules while writing programs. However, note that there has to be at least one whitespace character between each token, which has not been delineated by an operator or a separator. A whitespace can be a **space**, a **tab** or a **newline**.

2.2.3 Separators

Java uses the following separators :

;	to terminate a statement
,	to separate consecutive identifiers in declaring variables. Also used inside a **for** statement to chain statements
.	to separate package names from subpackages and classes, to separate a variable or a method from a reference variable.
()	used to contain a list of parameters in a method definition as well as invocation. Also used for defining precedence in expressions, and surrounding cast types. Also used for containing expressions in control statements.
{ }	used to define blocks of code, for classes, methods

and local scopes, also used to contain values of automatically initialised arrays.

[ ] used to declare array types. Also used when dereferencing array values.

2.2 Check Your Progress.

1. Write True or False

a) Java supports three types of comments.

b) In Java, a space is not treated as a whitespace.

c) , is a type of separator in Java.

2.3 IDENTIFIERS, LITERALS, AND KEYWORDS

2.3.1 Identifiers

Identifiers are names given to classes, methods, variables. An identifier can be a combination of uppercase and lowercase characters, numbers, the underscore and the dollar sign. But they cannot begin with a number. Also remember that Java is case sensitive so MAX is different from max.

BIG	THREE_one	count	a2	$first	are all valid identifiers
2a	max-val	first/last			are not valid identifiers

2.3.2 Literals

A literal is a **constant** value. Following are some examples of literals:

100 65.42 ‘A’ “string literal’

As you can see the first is an **integer** literal, the second a **float**ing point literal, the third is a **character** constant and the last is a **string**.

Integer Literals : Any whole number value is an integer literal. The bases that can be used for integer literals are decimal, octal and hexadecimal. Octal values in Java are denoted by a leading zero. Integer literals create an **int** type value which the Java compiler stores as a 32 bit value.

Floating point literals : These represent decimal values with a fractional component. They can be expressed either in the standard or in the scientific notation. By default floating literals are stored with double precision.

Boolean literals : boolean literals are only two logical values **true** or **false**. The values **true** and **false** do not convert to any numerical representation.

Character Literals : Character literals are 16 bit values which can be converted into integers and manipulated with the integer operators, like the addition and subtraction operators. A character literal is represented inside a pair of single quotes eg. ‘p’.

String Literals : These are specified by enclosing them in a pair of double quotes as shown in the example above. Remember that in Java string must begin and end on the same line, there is no line continuation escape sequence.

2.3.3 Keywords

Java has 48 keywords which when combined with the syntax of operators and separators form the definition of the Java language. Note that these keywords cannot be used as variable, method or class names. The keywords are :

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

The keywords **const** and **goto** have been defined but are not used. In addition, Java reserves : **true**, **false** and **null**. These are values defined by Java and these words cannot be used as names for variables, methods, classes etc.

The Java environment depends on a number of built in class libraries which contain numerous built in methods that provide support for input/output, string handling, networking etc. Thus Java in totality is a combination of the Java language itself, and its standard classes. The class libraries provide much of the functionality that comes with Java. So it is essential for the student to learn using the standard Java classes.

2.3 Check Your Progress.

1. Answer the following.

a) List the various types of literals supported by Java.

____

____

b) List the reserved words besides the keywords in Java.

____

____

2.4 DATA TYPES

Java defines the following simple data types :

Integers : This group includes **byte**, **short**, **int** and **long**. These are used for whole values which are signed numbers.

Floating point types : This group includes **float** and **double**. These represent values with fractional parts.

Characters : This group includes **char**, and they represent symbols in a character set. eg. letters, numbers.

Boolean : This is a special type for representing true/false values.

These simple data types form the basis of all the other types of data that you can create. It is important to note that these simple types represent single values and not complex objects. They are similar to the simple types found in most other non-object oriented languages. In Java all data types have a strictly defined range. This is essential to achieve the requirement of portability. This helps to make the programs architecture independant.

2.4.1 Integers

All the four integer data types viz. **byte**, **short**, **int** and **long** are **signed** i.e. positive and negative values. Java does not support unsigned integers. This differs from C/C++ where both signed and unsigned integers are supported.

byte : The smallest integer type is **byte**. This is a signed 8 bit type with a range from -128 to 127. A byte variable is declared using the keyword **byte**. eg.

```
byte b, b1;
```

Such types of variables are useful when working with a stream of data from a network or a file and also when working with raw binary data which may not be directly compatible with other built in types supported by Java.

short : short is signed 16-bit type which has a range of values form -32,768 to 32,767. eg.

```
short s, s1;
```

int : This is the most commonly used integer data type with a range of values from -2,147,483,648 to 2,147,483,647. It is a signed 32 bit data type, Variables of type **int** are mainly used to control loops and to index arrays. It is important to note that anytime you have an integer expression which involves bytes, shorts and ints, and literal numbers, the entire expression will be promoted to the type **int** before the calculation is done. **int** is the most versatile and efficient data type among integers.

long : long is signed 64 bit and useful in situations where the **int** data type is not large enough to hold a particular value.

2.4.2 Floating point types

Floating point numbers are also known as real numbers. These are useful in expressions involving fractional values. There are two types of floating point types : **float** and **double**.

float : float represents single precision value. It uses 32 bits of storage. It can be useful when you need a fractional component but where the degree of precision required is not very high. A **float** variable is declared with the **float** keyword. The following example will illustrate :

float max, min;

double : Double precision is denoted by the keyword **double**. It uses 64 bits to store a value. All transcendental mathematical functions like sin(), cos(), sqrt() etc. return **double** values. Double precision is actually faster than single precision on some modern processors of today which have been optimised for high speed mathematical calculations.

2.4.3 Characters

The data type used to store characters is **char**. Remember however that **char** in Java is different than C/C++ where the **char** is an integer which occupies 8 bits. In Java, Unicode is used to represented **char**. Unicode defines a fully international character set that can represent all of the characters found in all human languages. Therefore the **char** data type requires 16 bits and the range of **char** is from 0 to 65,536. There are no negative character data types. Even though **char** is not treated as integers, you can add chars or increment the value of char variables.

2.4.4 Boolean

boolean is the simple data type for logical values. It has only one of the possible values : **true** or **false**. This is the type returned by all relational operators. **boolean** is also the type required by the conditional expressions in the control statements like **if** and **for**.

2.4 Check Your Progress.

1. Write True or False

a) Integer data types are treated as objects in Java.

b) boolean is a simple data type.

c) In Java, the **char** data type is treated as integer.

2.5 VARIABLES

A variable is the basic unit of storage in a Java program. All variables must be declared before they are used. The general form of declaring a variable is :

type identifier [=value][,identifier[=value]....];

type is one of Java’s atomic type or the name of a class or an interface and identifier is the name of the variable. Variable can be initialised at the time of declaration by giving an equal to sign and a value. We use a comma to separate variables if we are declaring more than one variable of the specified type. Following examples illustrate :

```
int a = 10, b, c;
double radius;
char a = ‘a’, ch;
```

Java also allows variables to be initialised dynamically by using any expression valid at the time the variable is declared.

The following example will illustrate :

```
class Circum
{
                public static void main(String args[])
                {
                float r = 4.2;
                //initialise circumference dynamically
                float cir = 2 * 3.14 * r
                System.out.println(“Circumference is “+ cir);
                }
}
```

In this example you can see that the variable r is initialised with a constant value. Variable cir is dynamically initialised to the circumference of the circle with radius r.

Scope and Lifetime of variables : Java allows variables to be declared in any block. A **block** defines the **scope**. This means that everytime you create a new block, you create a new scope. The scope determines what objects are visible to other parts of the program and it also determines the lifetime of the objects. In Java, the two major scopes are those defined by a class and those defined by a method. The class scope has several unique properties and attributes. These do not apply to the scope defined by a method. Let us for the moment see the scope defined within a method.

Variables declared inside a scope are not visible to code that is defined outside the scope. This means that when a variable is declared within a scope, it is localised to that scope and is protected from unauthorised access and modification. Scope rules provide the basis of encapsulation. Scopes can also be nested. The objects declared in the outer scope will be visible to the code within the inner scope, however the objects declared within the inner scope are not visible outside it.

In Java, variables are created when their scope is entered and destroyed when their scope is left. Once a variable goes out of its scope, it will no longer retain its value. This means that the lifetime of a variable is only within its scope. If a variable is initialised at the time it is declared, then that variable will be reinitialised everytime the block in which it is declared is entered.

2.6 TYPE CONVERSION AND CASTING

Automatic type conversion : When one type of data is assigned to another type of variable, an automatic conversion takes place if the following conditions are satisfied:

- the two types are compatible

- the destination type is larger than the source type.

In such a situation, a **widening conversion** takes place. eg. when a **byte** is converted to **int**, automatic conversion will take place. In automatic conversions, the numeric data types including integers and floats are compatible with each other. However, the numeric data types are not compatible with **char** or **boolean**. **char** and **boolean** are not compatible with each other either. Java also performs automatic type conversion when storing a literal integer constant into variables of type **byte**, **short** or **long**.

Casting Incompatible types : In a situation like where you want to convert an **int** data type to a **byte**, automatic conversion is not possible since **byte** is smaller than **int**. In such situations you have to use a cast. A **cast** is an explicit type conversion. It takes the following form :

(*target-type*) value

the *target-type* specifies the desired type to convert the specified value to.

eg. int a;

byte b;

b = (byte) a;

In this example, an **int** is cast to a **byte**. Here if the value of the integer is bigger than the range of a byte, then it will be reduced modulo byte's range.

When a floating point value is assigned to an integer, a different type of conversion i.e. truncation will occur and the fractional component will be lost. Also, if the size of the whole number is large to fit within the target integer type, then the value will be reduced modulo the target type's range.

Automatic Type promotion : In Java, all **byte**s and **short**s are automatically promoted to **int**s. Also, if one operand is **long**, then the whole expression is promoted to **long**. If one operand is a **float** operand, the entire expression is promoted to **float** and if any of the operands is **double**, the result is **double**.

2.7 ARRAYS

An array is group of variables of the same type and which have a common name. Arrays can be of any type and can have one or more dimensions. Any specific value or element in an array is accessed by its index. Remember that arrays in Java differ from the way they work in C/C++.

One dimensional array : A one dimensional array is a list of like typed variables. The general form for creating a one dimensional array is :

type var-name[ ] ;

here *type* is the base type of the array which determines the data type of the elements of the array. *var-name* is an array variable. You can declare an array as shown below :

int arr1[ ];

here arr1 is an array type variable, but it is important to note that with this declaration no array actually exists. The value of arr1 is set to **null**, which represents an array with no value. To link arr1 to a list of variables of type **int**, you must allocate one using the **new** operator. **new** is a special operator that allocates memory. The general form of **new** when applied to a one dimensional array is :

array-var = new *type*[*size*];

type specifies the type of data, *size* specifies the number of elements in the array. *array-var* is the variable that is linked to the array. The elements in the array allocated by **new** will automatically be initialised to zero. Thus in the above example if we wish to allocate ten elements to arr1 then we can do so as :

arr1 = new int[10];

arr1 now refers to an array of 10 integers and the elements initialised to zero.

You can combine array declaration and allocation in one step as : int arr1 = new int[10];

An alternative form of declaring arrays is given below :

type [ ] var-name;

eg. int [ ] a1 = new int[5];

Here the square brackets come first after the type specifier and then the name of the array variable. This form is included mainly as a part of convenience.

All arrays are dynamically allocated in Java. Once you have allocated an array, you can access the elements of the array by specifying an index in a square bracket. All array indexes start at zero. i.e. arr1[0] is the first element of arr1, arr1[1] is the second element and so on.

Arrays can be initialised at the time of their declaration. An array initialiser is a list of comma separated expressions surrounded by curly braces. The commas separate the values of the array elements. Java will automatically create an array large enough to hold the number of values specified in the array initialiser. There is no need of using **new** when initialising arrays at the time of declaration.

eg. int odd_nos = {1, 3, 5, 7, 9, 11};

Java run time system will check to be sure that all array indexes are in the correct range, it makes sure that you do not accidently try to store or reference values outside the range of the array. This is different than in C/C++ where there is no bounds checking.

Multidimensional arrays :

Multidimensional arrays are arrays of arrays. In order to declare a multidimensional array variable you should specify each additional index using another set of square brackets. eg. if you wish to initialise a two dimensional array you can do so as follows :

int array1[][] = new int[4][5];

About pointers : Java does not support or allow pointers. This is because if it allows the use of pointers, Java applets would be allowed to breach the firewall between the Java execution environment and the host computer. Java is designed in such a way that as long as you stay within the confines of the execution environment, you will never need to use a pointer nor would there be any benefit in using one.

2.5 to 2.7 Check Your Progress.

1. Write True or False

a) Objects declared in the outer scope will be visible to the code within the inner scope.

b) Variables can be dynamically initialised in Java.

c) In Java, bytes and shorts cannot be automatically promoted to int.

d) When a floating point value is assigned to an integer, truncation occurs.

e) All arrays are dynamically allocated in Java.

2.8 OPERATORS

Most of the operators in Java can be divided into the following groups : arithmetic, bitwise, relational and logical. Java also supports certain additional operators to handle special situations. Most operators in Java work in the same way as they do in C/C++. Let us now study these operators in detail.

2.8.1 Arithmetic Operators

These are useful in mathematical operations. The opeands of arithmetic operators must be of numeric type. Arithmetic operators cannot be used on **boolean** types, but they can be used on **char** types. The following is the list of operators supported by Java :

Remember that the unary - operator negates the operand. When the division operator is applied to integer types, the fractional component of the result will be truncated. Also the modulus operator returns the remainder of the division operation. It can be applied to both integer and floating point types. However, in C it can be applied on to integer data types.

Operator	**Result**	**Operator**	**Result**
+	Addition	- (also unary -)	Subtraction
*	Multiplication	/	Division
++	Increment	--	Decrement
+=	Addition assignment	-=	Subtraction assignment
*=	Multiplication assignment	/=	Division assignment
%	Modulus	%=	Modulus assignment

The **arithmetic assignment operator** is used as follows :

eg. a = a + 5;   can be written as

a += 5;

Here we have made use of the addition assignment operator. Both the statements perform the same operation. As we can see from the above table, there are assignment operators for all the arithmetic binary operators. The general form can therefore be expressed as :

var op= expression;

The assignment operators are shorthand for their equivalent long forms. They are also implemented more efficiently by the Java run-time system than their equivalent long forms.

Increment and Decrement operators : As in C/C++ the increment operator increases the value of the operand by 1, and the decrement operator decreases the value of the operand by 1. Both are unary operators and can appear in both the postfix and prefix forms.

eg. a = a + 1;

can be rewritten using the increment operator as :

a++;

or

++a;

Similarly with the decrement operator.

2.8.2 Bitwise Operators

Bitwise operators can be applied to the integer data types, **byte**, **char**, **short**, **int** and **long**. They act upon invidisual bits of the operands. As we have already studied all the integer types except **char** are signed integers. This means they can represent positive and negative values. Java uses the **2s complement** encoding method. This means that the negative numbers are represented by inverting all the bits in the value and then adding 1 to the result. Remember that the high-order bit determines the sign of an integer.

Bitwise Logical operators : The bitwise logical operators are & (AND) , **|** (OR) , ^ (XOR) and ~ (NOT). The truth table of the bitwise logical operators is given below :

Bitwise NOT (~) is also called as the bitwise complement operator and it inverts all of the bits of its operand.

A	**B**	**A&B**	**A|B**	**A^B**	**~A**
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Bitwise AND (&) produces a 1 bit if both the operands are also 1 and a 0 bit otherwise.

Bitwise OR (|) : In bitwise OR if either of the bits in the operands is a 1, then the resultant bit is 1.

Bitwise XOR (^) : This operator combines the bits such that if exactly one operand is 1, then the result is a 1, otherwise the result is a 0. In bitwise XOR, whenever the second operand has a 0 bit, the first operand is unchanged.

The Left Shift : The left shift operator (**<<**) shifts all of the bits in a value to the left a specified number of times. The general form of the left shift operator is :

value << *num*

where *num* specifies the number of positions to left shift the value of *value*. For each shift left, the high order bit is shifted out and is lost, and a zero is brought in on the right. Shifting left by a value of 1, has the same effect as doubling the original value, therefore this can be used as an alternative way to multiplication by 2.

The Right Shift : This operator shifts all of the bits in a value to the right the specified number of times. The general form of right shift is :

value >> *num*

Where *num* specifies the number of positions to right shift the value in *value*. When a value has bits that are shifted off, those bits are lost. Each time a value is shifted to the right, it gets divided by two and discards any remainder. This means that right shift can be used for integer division by 2. When shifting to the right, the leftmost bits exposed by the right shift are filled in with the previous contents of the top bit. This is called as sign extension and it preserves the sign of the negative numbers when you right shift.

The Unsigned Right Shift : In the right shift operator, the sign of the numbers is preserved. However in certain situations, especially when you are not shifting numeric values you may not want the sign extension to take place. You would generally want to shift a zero in the high order bit irrespective of what its initial value is. For this purpose you make use of the unsigned right shift operator >>> which always shifts zeroes into the high order bit. This is known as unsigned shift.

Bitwise Operator Assignments : All the bitwise operators have a shorthand form which is similar to that of algeabric operators. It combines the assignment with the bitwise operation eg.

a = a << 2; can also be written as

a <<= 2;

x = x & y; can also be written as
x &=y;

2.8.3 Relational Operators

Relational operators determine the relationship that one operand has to the other operand. The relational operators in Java are :

The result of these operations yields a **boolean** value. Any type in Java, including integers, floats, chars, and booleans can be compared for equality and inequality. Only integer, floating point and character operands may be compared to see which is less than or greater than the other.

Operator	**Result**	**Operator**	**Result**
==	Equal to	!=	Not equal to
>	Greater than	>=	Greater than or equal to
<	Less than	<=	Less than or equal to

Java does not define **true** or **false** in the same way as C/C++. In C/C++, **true** implies any non zero value and **false** is a zero value. However, in Java, **true** and **false** are non numeric values. Therefore to test for zero or non zero you must explicit**y make use of one or more relational operators. The following example will illustrate :

In C/C++ you can write a block of code as :

```
int check;
if(check) ....
if(!check) ...
```

However, in Java you would be required to write as follows :

```
if(check == 0) ....
if(check != 0) ...
```

This means in Java to test for zero or non zero you have to explicitly make use of one or more of the relational operators.

2.8.4 Boolean Logical Operators

These operators operate only on boolean operands. All the binary logical operators combine two boolean values to form a resultant boolean value.

Operator	**Result**	**Operator**	**Result**
&	Logical AND	|	Logical OR
^	Logical XOR	||	Short circuit OR
&&	Short circuit AND	!	Logical unary NOT
&=	AND assignment	|=	OR assignment
^=	XOR assignment	==	Equal to
!=	Not equal to	?:	Ternary if-then-else

The logical Boolean operators &, | and ^ operate on boolean values in the same way that they operate on bits of an integer. The logical ! operator inverts the Boolean state, i.e. !true = false and !false = true.

Short Circuit Logical Operators : Short circuit operators are secondary versions of Boolean AND and OR operators. They are not found in most other computer languages. We can see from the truth table of the operators that the OR operator results in **true** when A is **true**, irrespective of whatever B is. Similarly the AND operator results in a **false** when A is **false**, irrespective of what B is. So if you use the short circuit OR (||) and short circuit AND (&&) forms rather than the | and & forms, Java does not evaluate the right hand operand because the result can be determined by the left hand operand alone.

2.8.5 The Assignment Operator

The assignment operator is the = sign. The general form of the assignment operator is :

var = *expression*;

Note that the type of *var* must be compatible with *expression*. Assignment operator allows you to create a chain of assignments which is an easy wasy to set a group of variables to a common value. eg.

int a, b, c;
a = b = c = 10;   is valid in Java.

2.8.6 The ?: operator

This operator works in the same way as it does in C/C++. The general form of this operator is :

exp1? *exp2* :: *exp3*

exp1 is any expression that evaluates to a boolean value. If *exp1* is *true* then *exp2* is evaluated else *exp3* is evaluated. The result of the operation is that of the expression that is evaluated (either *exp2* or *exp3*). Note that both *exp2* and *exp3* should return the same type and the return type cannot be **void**.

2.8.7 Operator Precedence

The following table shows the order of precedence of the Java operators from highest to lowest. Remember that the square brackets are

Highest			
()	[ ]		
++	--	~	!
*	/	%	
+	-		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		
Lowest			

2.8 Check Your Progress.

1. Fill in the blanks.

a) Bitwise OR is a __________ operator.

b) The __________ operator can be used as an alternative way to multiplication by 2.

c) Boolean operators operate only on __________ operands.

d) & has a ________ priority as compared to &&.

used for array indexing. The dot operator is used for dereferencing objects. Parenthesis are used to alter the precedence of an operation. Remember that using parenthesis do not negatively affect the performance of your program.

2.9 SUMMARY

We learn some java keywords, identifiers, literals. Literals are integer, floating point, Boolean, character and string.

Different datatypes used in java are int, byte, short, long, float, double, Boolean, char and string. Declaration of variables is

type identifier[=value][,identifier[=value]..]

Array: Array is group of variables of the same type and which have common name. Arrays can be ant type and can have one or more dimensions.

type var-name[] = new type[size];

Java does not support or allow pointers.

Most of the operators in java can be divided into the following groups : arithmetic, bitwise, rational and logical.

Source : *www.java-samples.com(Link)*

2.10 CHECK YOUR PROGRESS - ANSWERS

2.1

1. a) .java

b) .class

2. a) True

b) True

c) False

2.2

1. a) True

b) False

c) True

2.3

1. a) Integer literals, floating point literals, boolean literals, character literals and string literals.

b) true, false and null.

2.4

1. a) False

b) True

c) False

2.5

1. a) True

b) True

c) False

d) True

e) True

2.6

1. a) logical

b) left shift

c) boolean

d) higher

2.11 QUESTIONS FOR SELF - STUDY

1. Explain with example how you will compile and run a Java program.

2. Write brief notes on :

a) Comments

b) White spaces

c) Identifiers

d) The unsigned right shift operator

e) Short circuit operators in Java

f) Character Data Types

3. Write short notes on :

a) Separators in Java

b) Integer Data Types

c) Float Data Types

d) Scope and Life of variables.

e) Bitwise Logical operatos.

4. What do you mean by type conversion? What is widening conversion? Explain automatic type promotion.

5. What is an array in Java? Describe how arrays can be initialised in Java.

2.12 SUGGESTED READINGS

1. www.java.com
2. www.freejavaguide.com
3. www.java-made-easy.com
4. www.tutorialspoint.com
5. www.roseindia.net

Notes

Chapter : 3

Control Statements

3.0 OBJECTIVES

Dear Students,

After studying this chapter you will able to :

- explain the catagories of control statements in Java
- discuss selection statements **if** and **switch**
- discuss iteration statements **while**, **do-while** and **for**
- discuss the **break**, **continue** and **return** statements.

3.1 INTRODUCTION

The control statements in Java can be divided into the following catagories : **selection**, **iteration** and **jump**. Selection statements allow the program to choose different paths of execution based upon the outcome of an expression or the state of a variable. Iteration statements are useful to repeat one or more statements i.e. they form loops, and jump statements allow the program to execute in a non linear fashion. Control statements in Java are nearly identical to those in C/C++, there are only a few differences. Therefore the control statements shall be discussed very briefly in this section. In this chapter not many programming examples will be described for the various control statements of Java. It is left for the student to study these control statements in detail and apply them to the programming problems studied during the study of previous programming languages. However, a thorough study of these statements should be done.

3.2 SELECTION

The two selection statements in Java are : **if** and **switch**.

3.2.1 The if statement :

This is a **conditional branching statement**. The general form of **if** is :

```
if (condition) statement1;
else statement2;
```

Each statement may be a single statement or a group of statements enclosed within a pair of braces. The condition is an expression which returns a **boolean** value. Note that the **else** clause is optional. If the condition is **true**, then *statement1* is executed, else *statement2* is executed. Remember that both *statement1* and *statement2* will not be executed in any situation.

A **nested if** is an **if** statement within another **if** or **else**. Here it is important to remember that an **else** statement always refers to the nearest **if** statement which is within the same block and which is not already associated with an **else**.

A sequence of nested **if**s is the **if-else-if** ladder. The general form of the **if-else-if** ladder is :

```
if(condition)
  statement;
else if(condition)
      statement;
else if(condition)
      statement;
:
:
else
      statement;
```

The **if** statements are executed in a top down manner.

3.2.2 The switch statement :

The **switch** statement provides multiway branching and is often a better alternative to nested **if-else-if** statements. The general form of the switch statement is :

```
switch(expression) {
```

```
case value1:
                statement block;
                break;
case value2:
                statement block;
                break;
:
:
case valuen:
                statement block;
                break;
default:
                default statement block;
}
```

The expression must be of type **byte**, **short**, **int** or **char**.

Each of the values specified in the case statements must be of a type compatible with the expression. Each case value must be a unique literal, it cannot be a variable. Duplicate case values are also not allowed.

The value of the expression is compared with each literal value in the **case**. When a match is found, the set of statements following that **case** statement is executed. If none of the values match, then the default statement is executed. Note however, that the **default** statement is optional. If there are no case matches and there is no default statement then no further action takes place.

The use of the **break** statement is to terminate the statement block, when the **break** is encountered execution branches to the first line of the code that follows the **switch** statement.The **break** statement is optional. If you do not use the **break** statement, execution continues with the next **case**.

It is also possible to have nested **switch** statements. Each **switch** statement defines its own block so there are no conflicts between case constants in the inner **switch** and those in the outer **switch**.

Remember -

*The **switch** statement differs from **if** statement in that, the switch can test only for equality, whereas **if** can evaluate any type of **Boolean** expression.*

*No two case constants in the same **switch** can have identical values.*

*A **switch** statement is generally more efficient than a set of nested **if** statements.*

3.1 & 3.2 Check Your Progress.

1. Fill in the blanks.

a) The selection statements in Java are .... and ....

b) .... statements allow the program to execute in a non linear fashion.

c) The .... statement provides multiway branching.

d) Each case value is a switch statement must be a unique ....

3.3 ITERATION STATEMENTS

We have already studied that a loop executes a set of instructions repeatedly until a termination condition is met. In this section let us study the looping statements provided in Java.

3.3.1 while loop : The general form of the **while** is:

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

The condition can be any **Boolean** expression. The body of the loop is executed as long as the condition is **true**. When the condition becomes **false**, the control passes to the next line which immediately follows the loop. If you wish to have more than one statements after the **while**, then they should be enclosed within a pair of braces. Since the **while** loop evaluates the conditional expression at the beginning of the loop i.e. at the top, the body of the loop may not execute even once if the condition is **false** the first time itself.

The body of the **while** (or any of the loops in Java) can be empty. This is because Java allows the **null** statement i.e. a statement which consists only of a semicolon. Thus it is possible to write short loops without bodies when the controlling expression itself can handle all the details itself. The following example will illustrate:

```
class EmptyLoop {
        public static void main(String args[ ]) {
                int i, j;
                i = 100;
                j = 200;
                while(++i < --j)        //find midvalue of i and j
                ;
// null statement
                System.out.println("Midpoint is : " + i);
        }
}
```

The output of the program is :

```
Midpoint is 50
```

3.3.2 do-while loop :

The general form of the **do-while** is :

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

Each iteration of the **do-while** loop first executes the body of the loop and then evaluates the conditional expression. If the expression evaluates to **true**, then the loop repeats otherwise it terminates. The condition must be a **Boolean** expression. Note that the **do-while** loop always executes its body at least once. This is because it checks the expression at the bottom of the loop. The **do-while** loop is used in situations where the loop termination test is to be done at the end of the loop rather than at the beginning. eg. the **do-while** loop is useful in menu selection, where the body of the menu loop should be executed at least once.

3.3.3 for loop : The general form of the **for** statement is :

```
for(initialisation;condition;iteration) {
   //body of the loop
}
```

The **for** loop operates as follows. When the loop first starts, the initialisation portion of the loop is executed. Generally, this portion contains the expression which sets the value of the loop control variable. (The loop control variable acts as a counter which controls the loop.) Note that the initialisation expression is executed only once. Next is the *condition*, which must be a **Boolean** expression. The condition generally tests the loop control varaible and if the expression is **true** then the body of the loop is executed. The iteration is executed next. It is usually an expression which increments or decrements the loop control variable. The loop then iterates, first evaluates the condition, then executes the loop body, then executes iteration expression again. This process continues till the controlling expression becomes **false**.

It is also possible to declare the variable that controls the **for** loop in the initialisation portion of for. eg. for(int i = 0; i <=10; i++)

Here i is declared in the **for** loop. Note however, that the scope of i will end as soon as the **for** statement ends. Outside the **for** loop the variable will not exist.

You can include more than one statement in the initialisation and iteration portion of the **for** loop. Each statement should be separated by a comma. This is particularly useful in situations where the **for** loop is controlled by the interaction of more than one variable. eg.

```
class ForLoop {
        public static void main(String args[ ]) {
                int a, b;
                for(a = 1, b = 6; a<b; a++,b--) {
                        System.out.println("a = " + a);
                        System.out.println("b = " + b);
                }
        }
}
```

Study the output of the above program.

The condition controlling the **for** loop can be any Boolean expression. eg.

```
boolean over = false;
for(int i = 1; !over; i++)  {
    if(...)
   over = true;
}
```

Here the **for** loop will continue to execute until the variable over is set to **true**, which will be when the **if** condition evaluates to **true**. Remember that it does not test the value of i.

Java also allows loops to be nested. One loop can be nested in another.

3.3 Check Your Progress.

1. Write True or False.

a) The body of a loop can be empty in Java.

b) The while statement is generally used in menu selection.

c) It is not possible to include more than one statement in the initialisation portion of the for loop.

3.4 JUMP STATEMENTS

Jump statements are used to transfer control to another part of the program. Java supports three jump statements : **break**, **continue** and **return**. Another way Java handles change of program flow execution is a mechanism called exception handling which shall be studied in later chapters.

3.4.1 break

The break statement has three uses : to break from the **switch** statement, to exit a loop and it can be used instead of the **goto** (The goto statement is best avoided). We have already seen the use of **break** in **switch**.

You can forcefully terminate a loop using the **break**, bypassing the conditional expression and any remaining code in the loop. When the **break** statement is encountered in a loop, the loop is terminated and program control is transferred to the statement following the loop. The **break** statement can be used with any of the loops in Java. When **break** is used in a set of nested loops, it will only break out of the innermost loop, the outer loop remains unaffected.

Using break as a Form of Goto : Java defines an extended form of the **break** statement which is called the **labeled break**. With this form of the **break**, it is possible to break out of one or more blocks of code. These parts of code need not be part of a loop or **switch**, they can be any block of code. You can also specify where the execution should resume.

The general form of the labeled break is :

break *label*;

label is the name of the label that identifies a block of code. When this form of code executes control is transferred out of the named block of code. You can use a labeled **break** to exit from a set of nested blocks. However, you cannot use a **break** to transfer control to a block of code that does not enclose the **break** statement. A label is any valid Java identifier followed by a colon. Once a block is labeled, you can use this label as the target of the **break** statement. This will cause execution to resume at the end of the labeled block. The following example will illustrate :

```
class BreakDemo {
public static void main(String args[ ]) {
                    boolean flag = true;
                    first: {
                      second:  {
                         third: {
                           System.out.println("Before break");
                           if(flag) break second;
                         System.out.println("Block breaks, hence this
                    will not execute");
                         }
                         System.out.println("This won't execute, since
                    break second");
                      }
                    System.out.println("Encountered after executing break
                    second");
                    }
  }
}
```

The output of the program is :

```
Before break
Encountered after executing break second
```

In the example there are three nested blocks. Each block has its own label. When the **break** statement is encountered, it will cause the execution to jump forward, past the end of the block labeled second. It will skip the print statements within third and second.

One of the most common uses of labeled **break** is to exit from nested loops. Remember that you cannot break to any label which has not been defined for an enclosing block.

3.4.2 continue :

Sometimes you may want to continue running a loop, but stop processing the remaining loop body for this particular iteration. In such a situation, you make use of the **continue** statement. In a **while** and **do-while** loop the **continue** statement causes control to be directly transferred to the conditional expression that controls the loop. In the **for** loop, the control first goes to the iteration portion of the **for** statement and then to the conditional expression. In all the three cases, any intermediate code is bypassed.

continue may also specify a label to describe which enclosing loop to continue.

3.4.3 return :

The **return** statement is used to explicitly return from a method. It causes the program control to transfer back to the caller of the method. At any time in a method, we can use the **return** to cause execution branch back to the caller of the method. The **return** statement immediately terminates the method in which it is executed. A detailed discussion of **return** is deferred till the study of methods. It has been introduced here since it is catagorised as a jump statement which causes program control to transfer back to the caller of the method.

3.4 Check Your Progress.

1. Match the following

Column A	**Column B**
a) return	i) used to exit a loop
b) break	ii) bypasses intermediate control and transfers control to the beginning of the loop
c) continue	iii) used to transfer control back to the caller of the method

3.5 SUMMARY

In this chapter we learn control statements used in java. Control statements in java can be divided into the following categories: selection, iteration and jump.

Selection statements are if-else and switch.

Iteration statements are while loop, do-while loop and for loop. Jump statements are break, continue and return.

Source : *www.java-samples.com(Link)*

3.6 CHECK YOUR PROGRESS - ANSWERS

3.1 & 3.2

1. a) if, switch

b) jump

c) switch

d) literal

3.3

1. a) True

b) False

c) False

3.4

1. a) - (iii) b) - (i) c) - (ii)

3.7 QUESTIONS FOR SELF - STUDY

1. Which are the conditional statements supported by Java? Explain them in detail.

2. What are the control statements in Java? What is the difference between the while and the do-while. Explain in detail the for statement and its features in Java.

3.What is meant by jump statements? Describe the labeled break statement. Also explain the use of return.

3.8 SUGGESTED READINGS

1. www.java.com
2. www.freejavaguide.com
3. www.java-made-easy.com
4. www.tutorialspoint.com
5. www.roseindia.net

Notes

Chapter : 4

Classes and Methods

4.0 OBJECTIVES

Dear Students,

After studying this chapter you will able to :

- explain classes and method in Java
- describe constructors
- discuss method overloading and over riding
- explain the use of the static and final keywords
- explain concepts of recursion, call by value, call by reference
- discuss application of nested and inner classes
- discuss inheritance in Java

4.1 INTRODUCTION

A **class** is a logical construct upon which the entire Java language is built. A class defines the shape and nature of an object. Any concept that you wish to implement in a Java program, must be encapsulated within a class. A class defines a new data type. This data type is then used to create objects of that type. Thus a class is a template for an object and an object is an instance of a class.

When you define a class, you specify the data it contains and the code that operates on that data.

The general form of declaring a class is :

```
class classname {
        type instance-variable1;
        type instance-variable2;
        ...
        ...
        type instance-variablen;
        type methodname1(parameter list) {
                body
        }
        type methodname2(parameter list) {
                body
        }
        .....
        ....
        type methodnamen(parameter list) {
                body
                }
}
```

The variables defined within a class are called *instance variables*. The code is contained within methods. The variables and the methods within a class are called as *members* of the class. It is the methods that determine how the variables of the class i.e. the class' data can be used.

Each instance of a class contains its own copy of the instance variables. This implies that the data for one object is separate and unique from the data of another object. Remember that Java classes do not need to have a **main()** method. You only specify a **main()** method in a class if that class is the starting point of your program. Also remember that applets do not have a **main()** method.

4.2 DECLARING A CLASS

Let us begin our study of classes and methods with the following example.

```
class Volume {
                int length;
                int breadth;
                int height;
}
```

This is a class with the name **Volume** which defines three instance variables length, breadth and height. We have now created a new data type called Volume. We now use this data type to create objects of type Volume. A class declaration only creates a template, it does not create any objects. To actually create a object of the type Volume you use a statement as follows :

```
Volume vol = new Volume();          //create an object vol of type Volume
```

Thus **vol** will be an object of **Volume** and a physical reality. Every object of the type Volume will contain its own copies of the instance variables length, breadth and height.

It is possible to create multiple objects of the same class. Remember that changes to the instance variables of one object will have no effect on the instance variables of another object.

You can create multiple objects for the above class Volume as:

```
Volume vol2 = new Volume();
Volume vol3 = new Volume();
```

The **new** operator dynamically allocates memory for an object and returns a reference to it. This reference is the memory address of the object allocated by **new**. Thus in Java all objects must be dynamically allocated. The advantage of dynamic allocation is that the program can create as many objects as required during program execution. It may also be possible, that due to memory not being available, **new** may not be able to allocate memory for an object. In such situations a run-time exception occurs.

The above method of declaring objects can also be declared in two steps as follows :

```
Volume vol1;    //declare a reference to object
vol1 = new Volume(); // allocate the object
```

Thus we can see that the **new** operator dynamically allocates memory to the object. Its general form can be written as :

clas-var = new *Classname*();

where *class-var* is a variable of type *classname*. Note that the class name followed by the pair of parenthesis specifies the constructor for the class. A **constructor** defines what occurs when a class is created. If no explicit constructor is specified, Java automatically supplies the default constructor. We shall study how to define our own constructors subsequently.

It is important to note here that Java's simple types like integers or characters are not implemented as objects, and therefore you do not need to use the **new** operator with them. Java treats objects differently than the simple types.

To access the instance variables we make use of the dot (.) operator. eg. to access the length variable of vol you and assign it a value 10, you can use the following statement:

```
vol.length = 10;
```

Having understood the basic concept, let us now write a program to determine the volume by making use of the above class.

```
class Volume {
    int length;
    int breadth;
    int height;
}
class Demo {
    public static void main(String args[ ]) {
            Volume vol = new Volume();
            int v;
            vol.length = 10;
            vol.breadth = 10;
            vol.height = 10;
            v = vol.length * vol.breadth * vol.height;
            System.out.println("Volume is :" + v);
    }
```

}

Remember that the **main()** method is in the class Demo. Therefore save the file with the filename Demo.java. When you compile this file, you will see that two **.class** files are created, one for class Volume and one for class Demo. Thus each class is put into its own **.class** file. You can also put each class in a separate file. To run the program, you execute the Demo class. The output would be :

```
Volume is 1000
```

Now consider the following example :

```
Volume vol1 = new Volume();
Volume vol2 = vol1;
```

In such a situation it is important to remember that both **vol1** and **vol2** refer to the same object. This means any changes made to **vol1** will affect the object to which **vol2** is refering because they are both the same object. Now a subsequent assignment to vol1 will unhook **vol1** from the original object. However, neither the object nor **vol2** will be affected. eg.

```
Volume vol1 = new Volume();
Volume vol2 = vol1;
....
vol1 = null;
```

In this case, vol1 has been set to **null** but vol2 still points to the original object.

This implies that when you assign one object reference variable to another object reference variable, you do not create a copy of the object, you are only making a copy of the reference.

4.1 & 4.2 Check Your Progress.

1. Write True or False

a) A class defines a new data type.

b) Java does not supply a default constructor.

c) In Java, all objects are dynamically allocated.

d) All classes in Java should have a main() method.

2. Fill in the blanks.

a) The .... operator dynamically allocates memory for an object.

b) A .... is a template for an object and an object is an.................. of a class.

4.3 METHODS

The code that operates on the data is contained in the methods of a class. The general form of a method is :

type name(*parameter-list*) {

body of the method

}

type specifies the data type returned by the method. This can be any valid data type including the class type that you create. If a method does not return a value, its return type must be declared **void**. *name* specifies the name of the method. The *parameter-list* is a sequence of type and identifier pairs separated by commas. Parameters are variables which receive values of the arguments passed to the method, when it is called. If a method has no parameters, the parameter list will be empty.

Remember that methods which have a return type other than **void**, should return a value to the calling routine using the **return** statement as follows :

return *value*;

where *value* is the value returned.

We know that we can use methods to access the instance variables defined by the class. Methods define the interface to most classes and therefore the internal data structures remain hidden. Let us modify the above program to add a method to compute the volume.

```
class Volume {
    int length;
    int breadth;
    int height;
void calvol() {          // method to display volume
            System.out.print(“Volume is :”);
            System.out.println(length * breadth * height);
    }
}
class Demo {
    public static void main(String args[ ]) {
            Volume vol = new Volume();
            int v;
            vol.length = 10;
            vol.breadth = 10;
            vol.height = 10;
            vol.calvol();              // call method to
                               display volume
    }
}
```

The output of the program would be :

```
Volume is : 1000
```

/**vol.calvol()** invokes the method **calvol()** on the object **vol**. When the method is executed the Java run time system transfers control to the code defined inside the volume. When the code is executed control is transferred back to the calling routine and execution continues from the following line of code. Remember that in the method **calvol()** the instance variables are accessed directly without the dot operator. This is because a method is always invoked relative to some object of its own class.

In the above example, we have declared the return type of our method to be **void** which means our method will not return any value to the calling routine. We can modify the above method by computing the volume in the method and making the method return the value to the calling routine. The modified method may be written as :

```
int calvol() {
                        return length * breadth * height;
}
```

In our calling routine we can receive this value in a variable as :

```
v = vol.calvol();
```

where v is a variable of type **int**. Alternatively the volume can directly be printed without an intermediate variable as:

```
System.out.println("Volume is :" + vol.calvol());
```

In this case, vol.calvol() will be called automatically and its value passed to **println()**.

Remember :

- the type of data returned by a method must be compatible with the return type specified by the method.

- The variable receiving the value returned by a method must also be compatible with the return type specified for the method.

Parameterised methods :-

Parameters allow a method to be generalised. A parameterised method can operate on a variety of data and can be used in a number of slightly different situations. A parameter is a variable defined by a method which receives a value when the method is called. An argument is a value that is passed to the method when it is invoked.

We can further modify the above program and add a method which will receive values from the invoking routine as parameters. These can then be used to compute the volume. Here is a revised version of the program where we add a method **setval()**.

```
class Volume
{
                        int length;
```

```
        int breadth;
        int height;
        int calvol()             // method to compute volume
        {
                return length * breadth * height;
        }
        void setval(int l, int b, int h)
        {
                length = l;
                breadth = b;
                height = h;
        }
}
class Demo
{
        public static void main(String args[ ])
        {
                Volume vol = new Volume();
                vol.setval(10,5,10);        // call set
                                            val to set values
int v = vol.calvol();               // call method to compute
                volume
                System.out.println("Volume is : " + v);
        }
}
```

Thus the method **setval()** is used to set the dimensions length, breadth and height.

4.3 Check Your Progress.

1. Fill in the blanks.

a) If a method is declared .... it will not return any value to the calling routine

b) An .... is a value that is passed to the method when it is invoked.

c) A .... is a variable defined by a method which receives a value when the method is called.

4.4 CONSTRUCTORS

4.4.1 Constructors

It can be time consuming to initialise all of the variables in a class each time an instance is created. Therefore Java allows objects to initialise themselves when they are created. This automatic initialisation is performed through the use of a constructor. A constructor initialises an object as soon as it is created. It has the same name as the name of the class in which it resides. Constructors do not have a return type, not even **void**. This is because the implicit return type of the class' constructor is the class type itself. Let us modify the above program by adding a constructor to it.

```
class Volume {
                        int length;
                        int breadth;
                        int height;
                        Volume() {
                              length = 10;
                              breadth = 5;
                              height = 10;
                        }
                        int calvol() {          // method to calculate volume
                              return length * breadth * height;
                        }
                              }
class Demo {
                        public static void main(String args[ ]) {
                              int v;
                              Volume vol = new Volume();
                              Volume vol2 = new Volume();
                              v = vol.calvol();          // call method to
                                                         compute volume
                              System.out.println("Volume is : " + v);
                              v = vol2.calvol();
                              System.out.println("Volume is : " + v);
                        }
}
```

The output of the program would be :

```
Volume is : 500
Volume is : 500
```

Both the objects **vol** and **vol1** were initialised by the **Volume()** constructor when they were created. The constructor gave the same set of values to vol and vol2 and therefore the volume of both vol and vol2 will be the same.

Thus when you allocate an object with the following general form :

class-var = new *classname*();

when a constructor is not explicitly defined for a class, Java creates a default constructor. However, once you define your own constructor the default constructor is no longer used.

4.4.2 Parameterised constructors :

We saw that the above constructor initialised all the objects with the same values. But actually what is needed in practice is that we should be able to set different initial values. For this purpose, we make use of parameterised constructors and each object gets initialised with the set of values specified in the parameters to its constructor. The following modification in the above program will illustrate :

```
class Volume {
    int length;
    int breadth;
    int height;
    Volume(int l, int b, int h) {
        length = l;
        breadth = b;
        height = h;
    }
    int calvol() {          // method to compute volume
        return length * breadth * height;
    }
        }
class Demo {
    public static void main(String args[ ]) {
        int v;
        Volume vol = new Volume(10,5,10);
        Volume vol2 = new Volume(5,5,5);
        v = vol.calvol();             // call method to
                                compute volume
        System.out.println("Volume is : " + v);
```

```
                v = vol2.calvol();
                System.out.println("Volume is : " + v);
            }
}
```

The output of the program would be :

```
Volume is : 500
Volume is : 125
```

Remember that the parameters are passed to the **Volume()** constructor when new creates the object. Thus you pass values to the object vol when you create it as : Volume vol = new Volume(10,5,10).

4.4.3 this keyword

this is a keyword that can be used inside any method to refer to the current object i.e **this** is always a reference to the object on which the method was invoked. This is useful when a method needs to refer to the object that invoked it.

4.4.4 Instance Variable hiding

In Java, it is illegal to declare two local variables with the same name within the same or enclosing scopes. However, you can have local variables, including formal parameters to methods which overlap with the names of the class' instance variables. But when a local variable has the same name as the instance variable, then the local variable hides the instance variable. We can use the **this** keyword to refer directly to the object and thus resolve the name space collisions that might occur between the instance variables and the local variables. eg. we can modify Volume() as follows :

```
Volume(int length, int breadth, int height) {
            this.length = length;
            this.breadth = breadth;
            this.height = height;
}
```

Here the parameter names are also length, breadth and height, so we can

4.4 & 4.5 Check Your Progress.

1. Write True or False.

a) Constructors always have a return type.
b) Java handles deallocation automatically.
c) In Java, you can declare two local variables with the same name within the same or enclosing scopes.
d) It is possible that the finalize() method may not be executed at all.

make use of the keyword **this** to access the instance variables which have the same names.

4.5 GARBAGE COLLECTION

In languages like C++, dynamically allocated objects must be manually released by making use of the **delete** operator. Java however handles deallocation automatically. The technique which accomplishes this task is called *garbage collection*. When no references to an object exist, that object is assumed to be no longer needed and the memory occupied by that object can be reclaimed. There is no explicit need to destroy objects. However, remember that garbage collection only occurs sporadically during your program execution. It will not occur just because one or more objects exist which are not used anymore.

finalize() method :

Sometimes an object may be required to perform some specific action when it is destroyed. eg. if it is holding some non-Java resource like a file handle then such resources should be freed before the object is destroyed. For this purpose, Java provides a mechanism called **finalization**. With finalization, you can define specific actions that will occur when an object is just about to be destroyed. You can define a **finalize()** method to specify those actions that must be performed before an object is destroyed. The Java runtime mechanism calls that method whenever it is about to recycle an object of that class.

The general form of the method is :

```
protected void finalize() {
        finalization code
}
```

The **protected** keyword prevents access to **finalize()** by code defined outside its class. **finalize()** is called prior to garbage collection. It is not called when an object goes out of scope. Thus you have no way of knowing when (or even if) the method will be executed. Therefore your program must not depend upon **finalize()** for normal program operation. It must have other means of releasing system resources used by the object.

4.6 OVERLOADING

4.6.1 Overloading Methods

When two or more methods in the same class share the same name, as long as their parameter declarations are different the methods are said to bo *overloaded* and the process is referred to as *method overloading*. Java supports overloading. It is one of the ways that Java implements the concept of polymorphism.

When an overloaded method is invoked, Java uses the type and/or number of arguments as a guide to determine which version of the overloaded method to call. When Java encounters a call to an overloaded method, it executes that version of the method whose parameters match the arguments used in the call.

Overloading allows related methods to be accessed by the use of a common name. Thus through the application of polymorphism, several names have been reduced to one. When methods are overloaded each method can perform any desired activity. There is no rule stating that overloaded methods must relate to one another. However, in practice you should only overload closely related operations.

The following example demonstrates overloading methods.

```
    class Area {
                        void area(int i) {
                             int a = i * i;
                             System.out.println("Area of a square is :" + a);
                        }
                        void area(int i, int j) {
                        int a1 = i * j;
                        System.out.println("Area of a rectangle is :" + a1);
    }
    void area(float r) {
                        double a2 = 3.14 * r * r;
                        System.out.println("Area of a circle is :" + a2);
    }
}
class Over {
    public static void main(String args[]) {
                        Area A = new Area();
                        A.area(10);
                        A.area((float)7.2);
                        A.area(10,5);
    }
}
```

In the above example area() is overloaded thrice, one which takes one integer parameter, second takes two integer parameters and the third takes one float parameter. When you call a method, Java looks for a match between the arguments used to call the method and the parameter list of the method. Upon finding a match, the corresponding method is invoked. Study the output of the above program.

4.6.2 Overloading Constructors

It is also possible to overload constructors. The overloaded constructor is called based upon the parameters specified when **new** is executed at the time of creating objects. Let us write a program to demonstrate constructor overloading :

```
class Area {
        int length, breadth;
        Area(int i) {
                length = breadth = i;
        }
        Area(int i, int j) {
                length = i;
                breadth = j;
        }
        int area() {
                return length * breadth;
        }
}
class OverCons {
        public static void main(String args[]) {
                Area A = new Area(10,5);
                Area A1 = new Area(5);
                System.out.println(“The area of square is :” +
        A1.area());
                System.out.println(“The area of rectangle is :” +
        A.area());
        }
}
```

Here when the object A is intialised, the constructor with two parameters is used and for initialising object A1, the constructor with a single parameter is used.

4.6 Check Your Progress.

1. Write True or False.

a) It is not possible to overload constructors in Java.

b) Method overloading is one of the ways that Java implements the concept of polymorphism.

c) As a rule overloaded methods must relate to one another.

4.7 USING OBJECTS AS PARAMETERS

Objects can be passed as parameters to methods. One of the most common use of passing objects as parameters involves constructors. Many times it may be required to construct a new object that is initially the same as an existing object. To do this, you simply define a constructor that takes an object as a parameter. The example below illustrates :

```
Class Area {
    int length, breadth;
    Area(Area a) {
                length = a.length;
                 breadth = a.breadth;
    }
    Area(int i, int j) {
                length = i;
                breadth = j;
    }
    int area() {
                return length * breadth;
    }
}
class ObjParam {
    public static void main(String args[]) {
                Area A = new Area(10,5);
                Area A1 = new Area(A);
                System.out.println("The area is :" + A1.area());
                System.out.println("The area is :" + A.area());
    }
}
```

When the first object A is allocated, the constructor which takes two parameters is invoked. When creating object A1, object A is passed as parameter, and is used to initialise instance variables of object A1.

4.8 CALL BY VALUE AND CALL BY REFERENCE

The call by value method copies the value of an argument into the formal parameter of the subroutine. Therefore changes made to the parameters of the subroutine have no effect on the argument used to call it. In the call by reference method, a reference to an argument is passed to the parameter. This reference is then used in the subroutine to access the actual argument specified in the call.

This implies that any changes made to the parameter will affect the argument used to call the subroutine. Both call by value and call by reference are used in Java.

In Java, when you pass a simple type to a method, then it is passed by value. Thus changes made to parameters in the method, have no effect on the values of the argument. However, objects are passed as reference. We know that when we create a variable of a class type we are only creating a reference to an object. Therefore when you pass this reference to the method, the parameter that receives it will refer to the same object as that referred to by the argument. Changes made to objects within methods, affect the object that has been used as an argument. The following example will illustrate:

```
                Class One {
                      int a,b;
                      One(int i, int j) {
                               a = i;
                               b = j;
                      }
                      void method1(One o) {
                               o.a *= 2;
                               o.b /= 2;
                      }
                }
                class CallRef {
                      public static void main(String args[ ]) {
                               One O = new One(10,20);
System.out.println(“Values of O.a and O.b before calling method :” + O.a + O.b);
                               O.method1(O);
System.out.println(“Values of O.a and O.b after calling method :” + O.a + O.b);
                      }
                }
```

The output of the program will be :

4.7 & 4.8 Check Your Progress.

1. Write True or False.

a) Objects can be passed as parameters to methods.

b) In Java, simple types are passed to a method by reference.

c) Changes made to objects within methods, do not affect the object that has been used as an argument.

```
Values of O.a and O.b before calling method : 10, 20
Values of O.a and O.b before calling method : 20, 10
```

Thus you can see that with call by reference changes made to parameters, have affected the arguments.

4.9 RETURNING OBJECTS

A method can return any type of data including the class types that you created. The following example will illustrate :

```
Class One {
        int a,b;
        One(int i, int j) {
            a = i;
            b = j;
        }
        One method1() {
            One temp = new One(a,b)
            temp.a *= 2;
            temp.b /= 2;
            return temp;
        }
}
class RetObj {
        public static void main(String args[ ]) {
            One O = new One(10,20);
            System.out.println("Values of a and b for
        object O :" + O.a + O.b);
            One O1 = O.method1();
    System.out.println("Values of  a and  b for Object O1 :" +
O1.a + O1.b);
        }
}
```

The output of the program will be :

```
Values of O.a and O.b before calling method : 10, 20
Values of O.a and O.b before calling method : 20, 10
```

Remember that an object will not go out of scope because the method in which it was called terminates. It will continue to exist as long as there is a reference to it somewhere in the program. When there are no references, the object will be reclaimed next time garbage collection takes place.

4.10 RECURSION

Java supports recursion. A method that calls itself is a *recursive* method. We have already studied recursion in our study of the C programming language. Recall that when writing recursive methods, you must have an **if** statement somewhere to force the method to return without the recursive call being executed. Otherwise the method will never return.

4.11 ACCESS SPECIFIERS

Encapsulation links the data with the code that manipulates the data. Encapsulation also provides another important attribute viz. access control. By introducing access control you can control what parts of the program can access the members of a class and thus prevent misuse. In this section we shall study access control as it applies to classes. An access specifier determines how a member can be accessed. The access specifiers in Java are : **public**, **private** and **protected**. **protected** applies only when inheritance is involved. Java also defines a default access level.

When a member of a class has the specifier **public** associated with it, then such a member can be accessed by any other code in your program. When a class member is specified as **private**, then that member can be accessed by the members of that class alone. Therefore the **main()** method is always declared as **public**, since it is called by code that is outside the program i.e. the Java run time system. When no access specifier is used, then by default the member of a class is **public** within its own package, but cannot be accessed outside the package. A **package** is a grouping of classes. We shall study more about packages later.

An access specifier precedes the rest of a member’s type specification. eg.

```
private int i;
public double a;
```

are both examples of declaring variables with their access specifiers. This means that the access specifier begins the declaration statement of the member.

4.12 STATIC KEYWORD

In some situations, it may be necessary to define a class member which will be used independant of any object of the class. To create such a member we make use of the keyword **static**. When a member is declared **static**, it can be accessed before any objects of its class are created and without reference to any object. Both methods and variables can be declared **static**. Thus you can understand why the **main()** method is declared **static**. **main()** is declared **static** because it must be called before any object exists.

Instance variables declared as **static** are essentially global variables. When objects of its class are declared, no copy of static variable is made. Instead all instances of the class share the same static variable.

Methods declared **static** have the following restrictions :

- they can only call other static methods

- they must access only static data, it is illegal to refer to any instance variable inside of a static method.

- they cannot refer to **this** or **super** in any way.

Outside of the class in which they are created static methods and variables can be used independantly, without use of any object. The general form is :

classname-method();

where classname is the name of the class in which the static method is declared.

4.9 to 4.11 Check Your Progress.

1. Write True or False.

a) Java supports recursion.

b) A method can return data of the class type that you created.

2. Match the following.

	Column A		**Column B**
a)	private	i)	applies when inheritance is involved
b)	public	ii)	can be accessed by the members of that class alone
c)	protected	iii)	can be accessed by any other code in your program

4.13 FINAL KEYWORD

When a variable is declared **final**, its contents cannot be modified. You must initialise a **final** variable at the time of its declaration. Usage of **final** is similar to **const** in C/C++. eg.

```
final PI = 3.141
final MAX = 100;
```

It is a common coding practice to choose all uppercase identifiers for variables declared as **final**.

A note about Arrays : In Java, arrays are implemented as objects. Arrays have a special attribute which is found in its **length** instance variable. The size of the array i.e. the number of elements that the array can hold, is found in the **length** instance variable. All arrays have this variable and it will always hold the size of the array. Remember that the value of length has nothing to do with the number of elements that are actually in use. It only reflects the number of elements that the array is designed to hold. The following program demonstrates how to determine the size of the array using the **length** instance variable.

```
class Length {
    public static void main(String args[]) {
                        int a1 = new int[10];
                        int a2 = {2, 4, 6, 8, 10, 12};
                        System.out.println(“length of a1 is “ + a1.length);
```

```
                System.out.println(“length of a2 is “ + a2.length);
        }
}
```

The output of the program will be :

```
length of a1 is 10
length of a 2 is 6
```

4.12 & 4.13 Check Your Progress.

1. **Answer the following.**

a) Why is the main() method declared static?

____

____

b) What happens when a variable is declared final?

____

____

c) What does the length instance variable tell about an array?

____

____

4.14 NESTED AND INNER CLASSES

It is possible to define a class within another class; such classes are known as nested classes. If class B is defined within class A, then B is known to A, but not known outside A. A nested class can have access to the members of the class in which it is nested (including private members). However, the enclosing class does not have access to the members of the nested class.

There are two types of nested classes : **static** and **non static**. Static classes have the **static** modifier applied to them, which implies that they must access members of its enclosing class through an object and not directly. Due to this limitation, static nested classes are used rarely.

The most important type of nested class is the **inner** class which is a non static nested class. It has access to all the members of its outer class and can refer to them directly. Thus an **inner** class is fully within the scope of its enclosing class.

Let us study the following illustration of **inner** class.

```
Class Outer {
        int outer_i = 100;
        void test() {
```

```
                    Inner inner = new Inner();
                    inner.display();
               }
               class Inner {
                    void display() {
                      System.out.println("Display member variable of
outer : " + outer_i);
                    }
               }
    }
    class InnerOuter {
               public static void main(String args[]) {
                    Outer outer = new Outer();
                    outer.test();
               }
    }
```

The output of the above program is :

```
Display member variable of outer : 100
```

You can see from the above example, that the inner class named **Inner** is defined within the scope of the outer class named **Outer**. **Inner** directly access the variable outer_x. The **display()** method is defined in the inner class and the outer class creates and instance of the inner class to access the method **display()** as **inner.display()**. Remember that no code outside of class **Outer** can access the class **Inner**. Also remember that an inner class can be defined within any block scope. It is not necessary that an inner class be enclosed within the scope of a class only.

4.14 Check Your Progress.

1. Write True or False

a) A nested class cannot have access to the members of the class in which it is nested.

b) An inner class can be defined within any block scope.

4.15 INHERITANCE

4.15.1 Inheritance a Class

We have already studied that using inheritance we can create a general class that defines traits common to a set of related items. This class can then be inherited by other more specific classes. Each of these classes adds those features that are unique to it. A class that is inherited is called the **superclass**.

The class which inherits another class is called a **subclass**. Thus a subclass is a specialised version of a superclass, which inherits all the instance variables and methods defined by the superclass, at the same time adding its own unique elements.

In order to inherit a class we make use of the keyword **extends**. The general form of extending a class is :

class *subclass-name* extends *superclass-name* {

body of the class

}

The following example will demonstrate how to inherit a class.

```
Class A {
        int x, y;
        void display() {
                System.out.println(“x and y “ + x + y);
        }
}
class B extends A {
        int z;
        void showz() {
                System.out.println(“z = “ + z);
        }
        void add() {
                System.out.println(“x+y+z= “ + (x + y + z));
        }
}
class Inherits {
        public static void main(String args[]) {
                A a = new A();
                B b = new B();
                a.x = 10;
                a.y = 5;
                System.out.println(“Contents of Superclass A :”);
                a.display();    // invoke display on object a
                b.x = 20;       //Subclass can access public
                        members of superclass
                b.y = 30;
                b.z = 10;
```

```
                    System.out.println("Contents of Subclass ");
                    b.display();            // invoke the display
                                            method on object b
                    b.showz();
                    System.out.println("Call add method from
                    Subclass :");
                    b.add();        // add method of subclass
                }
}
```

The output of the program will be :

```
Contents of Superclass A :
10                  5
Contents of Subclass
x and y : 20        30
z = 10
x+y+z= 60
```

Here you can see that the subclass B includes all the members of its superclass A. Therefore it can access the variables x and y. Note that class A though a superclass of B, is completely indepedant stand alone class.

Remember that you can only specify one superclass for any subclass that you create. Inheritance of multiple superclasses is not possible in Java. (This differs from C++). However, a subclass can be a superclass for another subclass. Thus you can create a hierarchy of inheritance in which a subclass becomes a superclass of another superclass. No class can be a superclass of itself. A superclass can be used to create any number of subclasses. Also remember that a member which has been declared **private** is not accessible by any code outside its class, including the subclasses.

4.15.2 Super

Whenever a subclass needs to refer to its immediate superclass, it can do so by using the keyword **super**. **super** has two general forms. The first one calls the constructor of the superclass. The second one is used to access a member of the superclass that has been hidden by a member of a subclass.

In the first form of **super**, a subclass can call a constructor method defined by its superclass by using the following form :

super(*parameter-list*);

The *parameter-list* specifies any parameters needed by the constructor in the superclass. Remember that **super()** must always be the first statement executed inside a subclass' constructor. The following example will illustrate this use of **super**

```
class A {
        int i;
        int j;
        A(int a, int b)
        {
                i = a;
                j = b;
        }
        ...
        ...
}
class B extends A {
        int k;
        B(int p, int q, int r) {
                super(p,q);
                k = r;
        }
        ...
}
```

Here the constructor for subclass B is called with parameters p, q, r. We make use of **super()** with parameters p and q which initialise i and j. The class A no longer initialises these values itself. Remember that constructors can be overloaded, therefore **super()** can be called using any form defined by the superclass. The one that matches the arguments in number and type will be executed.

An important thing to note is that **super()** always refers to the superclass immediately above the calling class. The general form of **super** in the second method is:

super.*member*

member can be either a method or an instance variable. This form of **super** is applicable when members of the subclass hide the members of the superclass which have the same name in both classes. **super** allows access to the member of the superclass which has the same name as a member of the subclass.
eg. if a variable i is defined in both the superclass and the subclass, then using **super.i** in the subclass, you can refer to the variable i of the superclass.

4.15.3 Multilevel Hierarchy

You can build hierarchies which contain as many levels of inheritance as you like. eg. if you have three classes A, B and C, then C can be a subclass of B, where B is a subclass of A. In this case, C inherits all the properties of A and B. Remember that all classes can be written in separate files and compiled separately.

In a class hierarchy, constructors are called in their order of derivation, from superclass to subclass. Also remember that **super()** must be the first statement executed in the subclass constructor.

4.15.1 to 4.15.3 Check Your Progress.

1. Fill in the blanks.

a) A class that is inherited is called the ....

b) We make use of the keyword .... to inherit a class.

c) In a class .... constructors are called in their order of derivation.

2. What are the two general forms of super?

....

....

4.15.4 Method Overriding

It may so happen that a method in a subclass has the same name and type as that of the superclass. In this situation, the method in the subclass is said to override the method in the superclass and the method defined by the superclass will be hidden. Overridden methods in Java are similar to virtual functions in C++. Let us illustrate method overriding with the following example :

```
class A {
        int i, j;
        A(int a, int b) {
            i = a;
            j = b;
        }
        void display() {
            System.out.println( "i = " + i + "j = " + j);
        }
}
class B extends A {
        int k;
        B(int a, int b, int c) {
            super(a,b);
            k = c;
        }
        void display() {
                System.out.println(“k = “ + k);
        }
}
class Override {
        public static void main(String args[ ]) {
                B b = new B(4, 10, 12);
                b.display();
        }
}
```

The output of this program will be :

```
k = 12
```

Here, when you invoke the method display() on an object of type B, the version of display() which has been defined in B will be used. This means the version of B will override the version declared in A.

If you wish to access the method of the superclass, you can do so by using the keyword **super**. Modify the above example as shown where you will precede the display() method of B with the word **super**.

```
class B extends A {
        int k;
        B(int a, int b, int c) {
                super(a, b);
                k = c;
        }
        void display() {
                super.display()  // call display of A
                System.out.println(“k = “ + k);
        }
}
```

The output of the modified program will be :

```
i = 4 j = 10
k = 12
```

super.display() calls the superclass version of display().

Remember that method overriding occurs only when the names and the type signatures of the two methods are identical. If they are not then the methods are simply overloaded.

4.15.5 Dynamic Method Dispatch

This is the mechanism by which a call to an overridden function is resolved at run time rather than compile time. This is how Java implements run time polymorphism. When a method that is overridden is called through a superclass reference, Java determines which version of that method to execute, based upon the object being referred to at the time the call occurs. Thus, this determination is made at run time. Thus, it is the type of the object being referred to and not the type of the reference variable, which determines the particular version of the method to be executed.

4.15.4 to 4.15.5 Check Your Progress.

1. Answer the following.

a) When does a method in the subclass override a method in the superclass?

____

____

b) What is meant by dynamic method dispatch?

____

____

4.15.6 Using Abstract Classes

Sometimes it is necessary to create a superclass that only defines a generalised form. This generalised form will be shared by all its subclasses and each subclass will fill in its own details. Such a class determines the nature of the methods that the subclass must implement. You can require that a certain method be overridden by subclasses by specifying the **abstract** modifier. The general form to declare an **abstract** method is :

abstract *type name*(*parameter-list*);

Note that no method body is present. Any class which contains one or more **abstract** methods must also be declared **abstract**. There can be no object of an **abstract** class. This means that an abstract class cannot be instantiated with the **new** operator. You cannot declare **abstract** constructors or **abstract static** methods. Any subclass of an abstract class must either implement all of the abstract methods of the superclass, or itself be declared **abstract**. A class is declared **abstract** by writing the **abstract** keyword before the class keyword at the beginning of the class declaration.

The following example will illustrate :

```
abstract class A {
                    abstract void meth();
                    void meth1() {
                         System.out.println("Method in abstract class A");
                    }
}
```

```
class B extends A {
        void meth() {
                System.out.println("Implement the abstract method of superclass A");
        }
}
class Abstracts {
        public static void main(String args[]) {
                B b = new B();
                b.meth ();
                b.meth1();
        }
}
```

The output of the program would be :

```
Implement the abstract method of superclass A
Method in abstract class A
```

Remember that you cannot create an instance of class A. The method **meth()** which is **abstract** in class A is provided a body in the subclass B.

4.15.7 final

We have already seen one use of **final** to create a named constant. Let us study two more uses of final in this section.

Using final to prevent overriding : To disallow a method from being overridden, specify the **final** modifier at the start of the declaration of the method. This means that methods declared as **final** cannot be overridden. Once a method has been declared **final** in a class, its subclass cannot override it. There can be no method with the same name and type in the subclass. A compile time error will occur in such situation.

Using final to prevent inheritance : If you want to prevent a class from being inherited you precede the class declaration with the word **final**. When a class it declared **final**, all of its methods are also implicitly declared **final**. It is illegal to declare a class as both **abstract** and **final**, because an abstract class is incomplete by itself and requires the subclass to provide implementation to its methods.

4.15.6 to 4.15.7 Check Your Progress.

1. Write True or False.

a) An abstract class can be instantiated with the new operator.

b) A class can be declared as both abstract and final.

2. List two uses of final.

____

____

4.16 THE OBEJCT CLASS

Object class is a special class in Java. All the other classes are subclasses of **Object**. **Object** is the superclass of all the classes. This means that a reference variable of type **Object** can refer to an object of any other class. Also remember that since arrays are implemented as classes, a variable of type **Object** can also refer to any array.

The **Object** class defines the following methods :

Method	**Purpose**
Object clone()	Creates a new object that is the same as the object being cloned
boolean Equals(Object object)	Determines whether one object is equal to another
void finalize()	called before an unusual object is recycled
Class getClass()	Obtains the class of an object at run time
int hashCode()	Returns the hash code associated with the invoking object
void notify()	Resumes execution of a thread waiting on the invoking object
void noifyAll()	Resumes execution of all threads waiting on the invoking object
String toString()	Returns a string that describes the object
void wait()	Waits on another thread of execution

Among these, the methods **getClass()**, **notify()**, **notifyAll()** are declared as **final**. Other methods may be overridden. We shall see all these methods in detail as we progress through the study notes.

4.16 Check Your Progress.

1. Fill in the blanks.

a) The .... class is the superclass of all the classes.

b) The getClass() method is declared as .... in object.

4.17 SUMMARY

A class is a logical construct which defines the shape and nature of an object.

The variables defined within a class are called instance variables. The code is contained within methods. The variables and the methods within a class are called as members of the class. The new operator dynamically allocates memory for an object and returns a reference to it.

Constructors : Java allows objects to initialise themselves when they are created. This automatic initialisation is performed through the use of a constructor. A constructor has the same name as the name of the class in which it resides. Constructors do not have a return type.

Garbage collection : Java handles deallocation automatically. The technique which accomplishes this task is called garbage collection.

finalize() method : You can define a finalize() method to specify those actions that must be performed before an object is destroyed.

The call by value method copies the value of an argument into the formal parameter of the subroutine. In the call by reference method, a reference to an argument is passed to the parameter.

Class declared with in a class called nested class.

There are two types of nested classes : static and non static.

4.18 CHECK YOUR PROGRESS - ANSWERS

4.1 & 4.2

1. a) True

b) False

c) True

d) False

2. a) new

b) class, instance

4.3

1. a) void

b) argument

c) parameter

4.4 & 4.5

1. a) False

b) True

c) False

d) True

4.6

1. a) False

b) True

c) False

4.7 & 4.8

1. a) True

b) False

c) False

4.9 to 4.11

1. a) True

b) True

2. a) ii)

b) iii)

c) (i)

4.12 & 4.13

1. a) main() method is declared as static because when a member is declared static, it can be accessed before any objects of its class are created and main() has to be called before any object exists.

b) When a variable is declared final its contents cannot be modified.

c) The size of the array i.e. the number of elements that the array can hold is found in the length instance variable of an array.

4.14

1. a) False

b) True

4.15.1 to 4.15.3

1. a) superclass

b) extends

c) hierarchy

2. The two general forms of super are :

super(*parameter-list*);

super.member

4.15.4 & 4.15.5

1. a) When a method in a subclass has the same name and type as that of the superclass the method in the subclass overrides the method in the superclass and the method defined by the superclass will be hidden.

b) Dynamic method dispatch is the mechanism by which Java implements run time polymorphism whereby a call to an overridden function is resolved at run time rather than compile time.

4.15.6 & 4.15.7

1. a) False

b) False

2. Two uses of final : to prevent overriding, to prevent inheritance.

4.16

1. a) object

b) final

4.19 QUESTIONS FOR SELF - STUDY

1. What is a class? Describe the general form of declaring a class.
2. What are methods? What is the general form of declaring a method?
3. **Write short notes on :**
   a) Constructors
   b) Parameterised methods
   c) Call by value and call by reference
   d) Access specifiers in Java
   e) The final keyword
   f) Method overriding
   g) Abstract class
4. What is overloading?
5. What do you understand by garbage collection? Describe in brief the finalize() method.
6. Describe the use of the static keyword in Java.
7. Write in brief about nested and inner classes.
8. Describe what is meant by inheritance. What do you understand by multilevel hierarchy?
9. What is meant by dynamic method dispatch?
10. What is the Object class? Write any two methods defined in the Object class.

4.20 SUGGESTED READINGS

1. www.java.com
2. www.freejavaguide.com
3. www.java-made-easy.com
4. www.tutorialspoint.com
5. www.roseindia.net

❑ ❑ ❑

Notes

Notes

Chapter : 5

Packages and Interfaces

5.0 OBJECTIVES

Dear Students,

After studying this chapter you will able to :

- Describe what is meant by a package
- Explain how to define a package
- State the way of importing packages in programs
- Describe what is an interface
- Explain how to define and implement an interface

5.1 INTRODUCTION

Packages are containers for classes which are used to keep the class name space compartmentalised. They are stored in a hierarchical manner and are explicitly imported into new class definitions. **Interfaces** can be used to specify a set of methods which can be implemented by one or more classes. The interface, itself does not define any implementation. An interface is similar to an abstract class, however a class can implement more than one interface. On the other hand, a class can only inherit a single superclass. Packages and interfaces are two of the basic components of a Java program.

In general a Java source file contains the following four internal parts :

- A single package statement (optional)

- Any number of import statements(optional)
- A single public class declaration (required)
- Any number of classes private to the package (optional)

Till now, we have only used the single **public** class declaration in all our programs. We shall see the remaining in this section.

5.2 PACKAGES

5.2.1 CLASSPATH

Before studying packages it is important to understand **CLASSPATH**. The specific location that the Java compiler will consider as the root of any package is controlled by **CLASSPATH**. Till this time, we have been storing all our classes in the same, unnamed default package. This allows you to compile the source code and run the Java interpreter on the compiled program by giving the name of the class on the command line. This works because the default current working directory is usually in the **CLASSPATH** evironmental variable. This variable is defined for the Java run time system by default.

Suppose you create a class **MyPack** in a package called pack. This means you have to create a directory called pack and put **MyPack.java** into this directory. pack should now be made the current directory to compile the program. The class file **MyPack.class** which will be created after compilation will also be stored in the pack directory. Now when you try to run **MyPack**, the Java interpreter reports an error. This is because the class is now stored in a package called **pack**. Thereofore you cannot refer to it only as **MyPack**. You should now refer to the class by enumerating its package hierarchy and separating the package by dots. Thus the class will now be called **MyPack.pack**. Now if you try using **MyPack.pack**, you will still receive an error message. This is because **CLASSPATH** sets the top of the class hierarchy. So there is no directory called pack, in the current working directory since you are working in pack itself.

To overcome this problem you have two alternatives :

i) change directories up one level and type java **MyPack.pack**

ii) add the top of your development class hierarchy to the **CLASSPATH** environment variable. Then you will be able to use java **MyPack.pack** from any directory and Java is able to find the correct **.class** file. eg. if you are working on source code in a directory called c:\student then you should set your **CLASSPATH** as :

```
.;C:\student;C:\java\classes
```

5.2.2 Defining Packages

You can define classes inside a package that are not accessible to the code outside the package. You can also define class members that are only exposed to other members of the same package. Thus the classes can have knowledge of each other, but it is not exposed outside the package.

To create a package, you should include a package command as the first statement in a Java source file. Any classes defined within that file will belong to the specified package.

The general form for the package statement is :

package *pkg*;

where *pkg* is the name of the package. eg.

package pack

will create a package named pack.

Java uses the file system directories to store packages. Therefore any **.class** files for any classes declared to be a part of **pack**, must be stored in a directory called **pack**. The directory name must match the package name.

More than one file can be included the same package statement. You can also create a hierarchy of packages. The general form of a multilevel package statement is :

package *pkg1*[.*pkg2 [.pkg3]*];

remember that the package hierarchy should be reflected in the file system of your Java development system. eg. package **java.awt.image**;

should be stored in j**ava\awt\image** on your Windows file system.

Let us create a small package with the following example :

```
package Pack;
class Student {
    String name;
    int roll;
    Student(String n, int r) {
            name = n;
            roll = r;
    }
    void display() {
            System.out.println("Name : " + name + "  Roll No. : " + roll);
    }
}
class Studrec {
    public static void main(String args[]) {
            Student S[] = new Student[3];
            S[0] = new Student("Vikram ", 20);
            S[1] = new Student("Jerry ",10);
            S[2] = new Student("Vishnu ",15);
```

```
                for(int i = 0; i<3; i++)
                        S[i].display();
        }
}
```

Save this file as **Studrec.java** and put in a directory called **Pack**. When you compile this file, make sure that the resulting **.class** file has also been stored in Pack. Then execute **Studrec** at the command line as follows :

```
java Pack.Studrec
```

Either set the **CLASSPATH** environment as described previously or go into the directory above Pack to execute the program. Remember that since **Studrec** is now a part of the package **Pack**, it cannot be executed by itself i.e. *a command line*

java Studrec

cannot be used.

5.2.1 & 5.2.2 Check Your Progress.

1. Write True or False.

a) A class can implement more than one interface.

b) You should include a package command as the first statement in a Java source file.

c) You can define classes inside a package that are not accessible to the code outside the package.

5.2.3 CATAGORIES OF VISIBILITY FOR CLASS MEMBERS WITH REFERENCE TO PACKAGES

Packages act as containers for classes and other subordinate packages. Java addresses four catagories of visibility for class members with reference to packages :

- subclasses in the same package
- non subclasses in the same package
- subclasses in different packages

	Private	No modifier	Protected	Public
Same Class	Yes	Yes	Yes	Yes
Same Package subclass	No	Yes	Yes	Yes
Same package non subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non subclass	No	No	No	Yes

- Classes that are neither in the same package nor subclass.

The following table summarises the class member access

Anything declared **public** can be accessed from anywhere. Anything declared **private** is not accessible outside of its class. When a member does not have any specific access specification, it is visible to all subclasses as well as to other classes in the same package. This is the default access. If you want to allow an element to be seen outside your current package but only to classes that are the direct subclasses of your class, you have to declare that element as **protected**.

5.2.4 Importing packages

All of the built in Java classes are stored in packages. In order to make visible certain classes or entire packages, Java makes use of the **import** statement. Once imported, a class can be referred to directly using only its name. In a Java source file, the **import** statements occur immediately following the package statement (if it exists) and before any class definitions. The general form of the import statement is :

import *pkg1*[.*pkg2*].(*classname*|*);

pkg1 is the name of the top level package, *pkg2* is the name of the subordinate package and is separated by a dot. Practically there is no limit to the depth of a package hierarchy.Then you can specify either an explicit *classname* or a star (*). A * indicates that the Java compiler should import the entire package. eg.

```
import java.util.Date;
import java.io.*;
```

All the standard Java classes included with Java are stored in a package called **java**. The basic language functions are stored in a package called **java.lang**. **java.lang** is implicitly imported by the compiler for all the programs because is provides a lot of functionality. This is equivalent to importing it as :

```
import java.lang.*;
```

If a class with the same name exists in two different packages which you import using the * form, then compiler does not issue any warning unless you try to use one of the classes. In that case a compile time error occurs and you have to explicitly name the class and its package name.

Wherever you use a class name you can use its fully qualified name which will include its full package hierarchy. eg.

```
import java.util.*;
class Example extends Date {
}
```

can be written without import as :

```
class Example extends java.util.Date {
}
```

Only those items within the package declared as **public** will be available to non subclasses in the importing code.

5.2.3 & 5.2.4 Check Your Progress.

1. Fill in the blanks.

a) The basic language functions in Java are stored in a package called ....

b) To make visible certain classes or entire packages, Java provides the .... statement.

c) To allow an element to be seen outside the current package, but only to classes that are the direct subclasses of your class, that element should be declared as ....

5.3 INTERFACES

5.3.1 Declaring an Interface

Using an interface you can specify what a class must do, but not how it does it. You can fully abstract a class' interface from its implementation. Interfaces do not have instance variables and their methods do not have any bodies. Once an interface is defined, any number of classes can implement an interface. Also one class can implement any number of interfaces. In order to implement an interface, a class must create a complete set of methods defined by the interface, but each class is free to implement its own details.

Interfaces are designed to support dynamic method resolution at run time. Usually, for a method to be called from one class to another class, both the classes need to be present at compile time. Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface.

The general form of an interface is :

```
access interface name {
        return-type method-name1(parameter-list);
        return-type method-name2(parameter-list);
        return-type method-name2(parameter-list);
        .....
        return-type method-nameN(parameter-list);
        type final-varnameN = value;
}
```

In this form, access is either **public** or not used. The default access applies when no specifier is used. It implies that the interface is only available to other members of the package in which it is declared. When the interface is declared **public**, the interface can be used by another code. *name* is the name of the interface which is a valid identifier. The methods in an interface are abstract methods. Each class which includes an interface must implement all the methods.

Variables can also be declared inside interface declarations. They are implicitly **final** and **static**. This means they cannot be changed by the classes which implement the interface. The variables must be initialised with constant values. If an interface is declared **public**, then all the methods and variables are implicitly declared **public**.

Let us see an example of interface declaration :

```
interface A {
      void meth1(int i);
}
```

5.3.1 Check Your Progress.

1. Fill in the blanks.

a) Variables declared in interfaces are implicitly .... and ....

b) The .... access applies when no specifier is used for an interface.

c) Interfaces do not have .... variables.

5.3.2 Implementing an interface

To implement an interface include the **implements** clause in the class definition and then create the methods that are defined by the interface The general form of a class which includes the implements clause is :

access class *classname* [extends *superclass*]

[implements *interface* [*,interface....*]] {

class body

}

access is either **public** or not used. If a class implements more than one interface they should be separated by commas. The methods that implement an interface must be declared **public**. Also note that classes which implement interfaces, can have additional members of their own also.

The following example will demonstrate how to implement the above declared interface :

```
class One implements A {
      public void meth1(int i) {
                  System.out.println(“Implement interface A in class One”);
                  System.out.println(“i is :” + i);
      }
}
```

You can also declare variables as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be stored in such a variable. When a method is called through any such references, the correct version based upon the actual instance of the interface being referred to will be called.The method to be executed is looked up dynamically at run time. This allows classes to be created later than the code which calls methods on them.

Let us call the method **meth1()** in the above example with the help of an interface reference variable as shown :

```
class Iface {
        public static void main(String args[]) {
                A Ainter = new  One();
                Ainter.meth1(10);
        }
}
```

In this example, the variable AInter is of interface type A, but it is assigned an instance of class One. Ainter can access meth1(), but it cannot access any other members of the class One because the interface reference variable has only knowledge of the methods declared by that interface declaration. It cannot be used to access any other methods of the class.

If a class includes an interface but does not fully implement the methods defined by that interface, then the class must be declared **abstract**. eg

```
abstract class Two implements A {
        int p, q;
        void display() {
                System.out.println("p and q" + p + q);
        }
}
```

This class does not implement meth1() of the interface A therefore is declared **abstract**.

Interfaces can be extended i.e. one interface can inherit another by the use of the keyword **extends**. When a class implements an interface which inherits another interface, then it must provide implementations for all the methods defined within the interface inheritance chain.

5.3.3 Variables in Interfaces

You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables which are initialised to the desired values. When the interface is included in a class all those variables names will be in scope as **constants**. (This is similar to a header file in C/C++ which contains a number of **#define** constants or **const** declarations).

5.4 SUMMARY

Packages are containers for classes which are used to keep the class name space compartmentalized. Packages act as containers for classes and other subordinate packages.

Interfaces can be used to specify a set of methods which can be implemented by one or more classes.

5.5 CHECK YOUR PROGRESS - ANSWERS

5.2.1 & 5.2.2

1. a) True

b) True

c) True

5.2.3 & 5.2.4

1. a) java.lang

b) import

c) protected

5.3.1

1. a) static

b) default

c) instance

5.3.2 & 5.3.3

1. a) False

b) True

2. a) implements

b) abstract

5.6 QUESTIONS FOR SELF - STUDY

1. What is meant by packages and interfaces?
2. Explain with example what is CLASSPATH.
3. Explain the general form of declaring a package.
4. What do you understand by public, private and protected access, with relation to packages?
5. Write notes on :
   a) Declaring an Interface
   b) Implementing an Interface

5.7 SUGGESTED READINGS

1. www.java.com
2. www.freejavaguide.com
3. www.java-made-easy.com
4. www.tutorialspoint.com
5. www.roseindia.net

Notes

Notes

Chapter : 6

Exception Handling

6.0 OBJECTIVES

Dear Students,

After studying this chapter you will be able to

- discuss what is an exception
- explain exception handling with the keywords try, catch, throw, throws and finally
- discuss the exception types
- explain how to use the try and catch clauses, multiple catch and nested try statements
- discuss the use of throw, throws and finally

6.1 INTRODUCTION

An exception in Java is an object that describes an exceptional i.e. error condition which has occured in a piece of code. When such an exception condition arises, an object representing that exception is created and thrown in the method that caused the error. The method may handle the exception on its own or may pass it on. In any case, at some point the exception gets caught and is processed.

Exceptions can be generated by the Java run time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors which violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions can be used to report error conditions to the caller of a method.

6.2 EXCEPTION HANDLING

Exception handling in Java is managed by five keywords : **try**, **catch**, **throw**, **throws**, and **finally**. A **try** block contains the program statements which you want to monitor for exceptions. If an exception occurs within the **try** block, then it is thrown. Using **catch**, your code can **catch** this exception and handle it in the appropriate manner. The exceptions generated by the system are automatically thrown by the Java run time system. The keyword **throw** is used to manually throw an exception. Any exception that is thrown out of a method is specified by the **throws** clause. Any code that must absolutely be executed before a method returns is put in a **finally** block.

The general form of an exception handling block is :

```
try {
        // try block to monitor for errors
}
catch(ExceptionType1 exOb) {
        // exception handler for ExceptionType1
}
catch(ExceptionType2 exOb) {
        // exception handler for ExceptionType2
}
// ....
finally {
        // block of code to be executed before try block ends
}
```

ExceptionType is the type of exception that has occured.

6.3 EXCEPTION TYPES

All exceptions are subclasses of the built in class **Throwable**. **Throwable** class is at the top of the exception hierarchy. Immediately below **Throwable** are two subclasses. These subclasses partition the exceptions in two distinct branches. One branch is **Exception**. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types. An important subclass of **Exception** is called

Runtime Exception. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero, invalid array indexing etc.

The other branch is topped by **Error**. This defines the exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of the type **Error** are used by the Java run time system to indicate errors which relate to the run time environment itself. eg. stack overflow. In this chapter we shall not deal with exceptions of type **Error**.

6.1 to 6.3 Check Your Progress.

1. Fill in the blanks.

a) All exceptions are subclasses of the class ....

b) Immediately below Throwable are two subclasses .... and ....

c) The keyword .... is used to manually throw an exception.

d) Any code that must absolutely be executed before a method returns is put in a .... block.

2. Answer the following.

a) What is an exception?

____

____

b) What do exceptions thrown by Java relate to?

____

____-

6.4 USING TRY AND CATCH

The default exception handler is provided by the Java run time system due to which exceptions get caught. However, you will usually want to handle an exception yourself. Handling exception by yourself will allow you to fix the error and at the same time prevent automatic program termination (which happens with the default exception handler).

In order to monitor a code for exceptions, enclose it in a **try** block and immediately following the **try** block, include a **catch** clause which specifies the exception type you wish to catch. The following example will illustrate :

```
class Excep {
    public static void main(String args[ ]) {
        int a, b;
        try {
                b = 0;
                a = 10/b;
            }
            catch(ArithmeticException e) {
                System.out.println(“Division by zero error”);
            }
                System.out.println(“After executing catch
        clause”);
        }
}
```

The output of the program will be :

```
Division by zero error
After executing catch clause
```

Remember that once an exception is thrown, control transfers out of the **try** block into the **catch** block. Once the **catch** clause is executed control continues with the next line in the program following the entire **try-catch**. Note that the scope of the **catch** clause is restricted to those statements specified by the immediately preceding **try** statement.The aim of most well constructed **catch** clauses should be to resolve the exceptional condition and then continue as if the exception never happened.

Displaying the Description of an Exception :

Throwable overrides the **toString()** method defined by **Object** and therefore it returns a string which contains the description of the exception. This description can be displayed in a **println()** statement by passing the exception as an argument. eg.The above example can be modified as :

```
catch(ArithmeticException e) {
        System.out.println(“Exception ” + e);
}
```

When the code is modified as above and a division by zero occurs the following will be displayed :

```
Exception : java.lang.ArithmeticException / by zero
```

6.5 MULTIPLE CATCH CLAUSES

In situations where more than one exception could be raised by a single piece of code, you can specify two or more **catch** clauses, where each will catch a different type of exception. When an exception is thrown, each **catch** statement is inspected in order and the first one whose type matches that of the exception is executed. No sooner does one **catch** statement execute, the others are bypassed and the control is transferred to the line following the try/catch block. The following example will demonstrate :

```
     class MultipleCatch {
            public static void main(String args[]) {
     try {
            int a = args.length;
            System.out.println("a= " + a);
            int b = 2/a;
            int c[] = {1};
            c[10] = 25;
     }
     catch(ArithmeticException e) {
            System.out.println("Division by zero : " + e);
     }
     catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index out of bounds " + e)
     }
     System.out.println("After try/catch block");
     }
}
```

If this program is started with no command line parameters it will cause the division by zero exception. If you provide a command line argument setting a to something greater than zero, then this exception will not occur. However, it will cause **ArrayOutofBoundsException** since int array c has length 1 and the program attempts to assign a value to c[10].

The output generated in both cases will be as shown below:

```
C:\> java MultipleCatch
a = 0
Divide by 0 : java.lang.ArithmeticException: / by zero
After try/catch block
C:\>java MultipleCatch Arg
a = 1
Array index out of bounds : java.lang.ArrayIndexOutOfBoundsException : 10
After try/catch block
```

Remember that when you use multiple **catch** statements the exception subclasses must come before any of their superclasses. This is because a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus a subclass would never be reached if it came after its superclass. Also, in Java unreachable code is an error.

eg. in a multiple catch clause suppose you first attempt to catch an **Exception** and then **ArithmeticException**, then the **catch** statement which catches the **ArithmeticException** will never reach. This is because **ArithmeticException** is a subclass of **Exception**. Therefore the first catch itself will handle all **Exception**-based errors including **ArithmeticException**. To correct this situation, you should first write a catch to tackle **ArithmeticException** and then **Exception**.

6.6 NESTED TRY STATEMENTS

A **try** statement can be enclosed inside the block of another **try**. Each time a **try** statement is entered, the context of its exception is pushed onto the stack. If an inner **try** statement does not have a **catch** for a particular exception, the stack is unwound and the next **try** statement's catch handlers are checked for a match. This continues till a corresponding **catch** is found or till all the nested **try** statements are exhausted. In a situation where no catch statement matches, the Java run-time system handles the exception.

The following example illustrates nested **try:**

```
class Nestedtry {
            public static void main(String args[]) {
                  try {
                        int a = args.length;
                        int b = 2/a;
                        System.out.println("a = " + a);
                        try {                  //nested try block
                              if(a != 0) {
                                    int c[ ] = {1};
                                    c[10] = a;
                  }
            } catch(ArrayIndexOutOfBoundsException e) {
                  System.out.println("Array index out of
      bounds :" + e);
            }
            } catch(ArithmeticException e) {
                  System.out.println("Division by zero : " + e);
            }
      }
}
```

The **try** block to check array index is nested within the try block which checks for division by zero. Thus when the program is executed without any command line arguments, the exception is generated by the outer try block. If the program is executed with one or more command line arguments then the exception is generated in inner try block. Check the output of this program as an exercise.

Remember that you can enclose a call to a method within a **try** block.

6.4 to 6.6 Check Your Progress.

1. Write True or False.

a) Once an exception is thrown, control transfers out of the try block into the catch block. ....

b) Try statements cannot be nested. ....

c) Throwable overrides the toString() method defined by Object. ....

d) Exception is subclass of Arithmetic Exception. ....

6.7 THROW

The **throw** statement allows your program to throw an exception explicitly. The general form of **throw** is :

throw *ThrowableInstance*;

ThrowableInstance must be an object of type **Throwable** or a subclass of **Throwable**. Simple types like **int** or **char**, as well as non-Throwable classes like **Strings** cannot be used as exceptions. A **Throwable** object can be obtained in two ways :

- using a parameter into a catch clause
- creating one with the **new** operator.

The flow of execution stops immediately after the **throw** statement, and subsequent statements are not executed. The nearest enclosing **try** block is checked to see whether it has a **catch** statement that matches the type of exception. If a match is found, control is transferred to that statement. If no match is found, the next enclosing **try** statement (if there is one) is inspected and so on. If no match is found, then the default exception handler halts the program. The following example shows the use of **throw** :

```
class ThrowDemo {
public static void main(String args[ ]) {
        try {
    try {
            throw new NullPointerException("Use of throw");
    }
    catch(NullPointerException e) {
            System.out.println("Caught Exception thrown by
throw");
            throw e;
```

```
                }
            }
            catch(NullPointerException e) {
                System.out.println("Exception caught : " + e);
            }
            }
        }
```

In this example **new** is used to construct an instance of **NullPointerException**. All of Java's built-in run-time exceptions have two constructors : one with no parameter and one that takes a string parameter. In the second form, the argument specifies a string which describes the exception. This string is displayed when the object is used as an argument to **println()**.

6.8 throws

A **throws** clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type **Error** or **RuntimeException**, or any of their subclasses. All other exceptions that a method can throw must be declared in the **throws** clause, if they are not a compile time error will occur.

The general form of the *throws* clause is :

type *method-name*(*parameter-list*) throws *exception-list* {

    // body of the method

}

exception-list is a comma separated list of the exceptions that a method can throw.

Study the following example.

```
class Demothrows {
        static void throwmeth() throws
IllegalAccessException {
                System.out.println("Inside throwmeth");
                thrownew IllegalAccessException("Demo");
        }
        public static void main(String args[ ]) {
                try {
                        throwmeth();
                }
                catch(IllegalAccessException e) {
                        System.out.println("Caught
        Exception " + e);
                }
        }
}
```

6.7 & 6.8 Check Your Progress.

1. Fill in the blanks.

a) The .... statement allows your program to throw an exception explicitly.

b) A .... clause lists the types of exceptions that a method might throw.

2. Write True or False.

a) Simple types like int or char can be used as exceptions ....

b) All of Java's built-in run-time exceptions have two constructors ....

6.9 finally

finally is used to create a block of code that will be executed after a try/catch block has completed and the before the code following the try/catch block. The **finally** block will be executed whether or not an exception is thrown. If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception. The **finally** clause is optional. However, each **try** statement requires at least one **catch** or a **finally** clause. The **finally** clause can be particularly useful for closing file handles, and freeing up other resources which might have been allocated at the beginning by a method.

6.10 Built in Exceptions in Java

The standard package **java.lang** defines several exception classes. The most general of these exceptions are subclasses of the standard type **RuntimeException**. **java.lang** is implicitly imported into all Java programs, therefore most of these exceptions are automatically available. These methods need not be included in any methods **throws** list either. Such exceptions are called as unchecked exceptions because the compiler does not check to see whether a method handles or throws these exceptions. Those exceptions which must be included in a method's **throws** list if the method can generate one of these exceptions and does not handle it itself.

Some Unchecked Runtime Exception Subclasses :

ArithmeticException

ArrayIndexOutOfBoundException

IllegalStateException

NegativeArraySizeException

IllegalMonitorStateException

Some Checked Exceptions defined in java.lang

ClassNotFoundException

IllegalAccessException

InterruptedException

NoSuchMethodException

Creating Exception Subclasses

In order to create your own exception types to handle situations specific to your application, define a subclass of **Exception**. (**Exception** is a subclass of **Throwable**). The **Exception** class does not define any methods of its own. It inherits those methods provided by **Throwable**. Thus all exceptions including those created by you have the methods defined by **Throwable** available to them.

The methods defined by **Throwable** include

Throwable fillINStackTrace()

String getLocalizedMessage()

String getMessage()

void printStackTrace()

6.9 & 6.10 Check Your Progress.

1. Write True or False.

a) The finally block will be executed even if an exception is not thrown.

b) The Exception class defines its own methods.

c) java.lang has to be explicitly imported into all Java programs constructors.

6.11 SUMMARY

An exception in Java is an object that describes an exceptional i.e. error condition which has occured in a piece of code. Exceptions can be generated by the Java run time system, or they can be manually generated by your code.

Exception handling in Java is managed by five keywords: try, catch, throw, throws, and finally. A try block contains the program statements which you want to monitor for exceptions. If an exception occurs within the try block, then it is thrown. Using catch, your code can catch this exception and handle it in the appropriate manner. The exceptions generated by the system are automatically thrown by the Java run time system. The keyword throw is used to manually throw an exception. Any exception that is thrown out of a method is specified by the throws clause. Any code that must absolutely be executed before a method returns is put in a finally block.

Source : *www.elearning.shadoze.com(Link)*

6.12 CHECK YOUR PROGRESS - ANSWERS

6.1 to 6.3

1. a) Throwable

b) Exception, Error

c) throw

d) finally

2. a) An exception in Java is an object that describes an exceptional i.e. error condition which has occured in a piece of code.

b) Exceptions thrown by Java relate to fundamental errors which violate the rules of the Java language or the constraints of the Java execution environment.

6.4 to 6.6

1. a) True

b) False

c) True

d) False

6.7 & 6.8

1. a) throw

b) throws

2. a) False

b) True

6.9 & 6.10

1. a) True b) False c) False

6.13 QUESTIONS FOR SELF - STUDY

1. What is an exception? List the five keywords used for exception handling in Java. Write and describe the general form of exception handling block.
2. Explain with an example how you will use the try and catch block.
3. Illustrate with example the nested try statement.
4. Describe the throw and throws statements using their general forms.
5. Write a short note on the keyword finally.

6.14 SUGGESTED READINGS

1. www.java.com
2. www.freejavaguide.com
3. www.java-made-easy.com
4. www.tutorialspoint.com
5. www.roseindia.net

❑ ❑ ❑

Notes

Chapter : 7

Multithreaded Programming

7.0 OBJECTIVES

Dear Students,

After studying this chapter you are able to

- discuss what is multithreaded programming
- explain the Thread class in Java
- explain to create a thread and multiple threads
- discribe the methods isAlive() and join()
- discuss how to set priority of threads
- to get an introduction of synchronisation
- to get a brief overview of interthread communication

7.1 INTRODUCTION

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

Thus, multithreading is a specialised form of multitasking. We have already studied that a process is a program in execution. This means that process based multitasking allows the computer to run two or more programs concurrently. In process based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler. In thread based multitasking environment, the thread is the smallest unit of dispatchable code. Therefore a single program can perform two or more tasks concurrently. Multitasking threads require less overhead as compared to multitasking processes. Processes are heavyweight and require their own separate address spaces. Threads on the other hand are lightweight. They share the same address space. Interthread communication is inexpensive as compared to interprocess communication which is expensive and limited.

Multithreading enables you to write very efficient programs which make maximum use of CPU, because idle time can be kept at a minimum. This is very important for the interactive, networked environment in which Java works.

Threads in Java exist in several states. A thread can be **running**. It can be **ready to run** as soon as it gets the CPU time. A thread which is running can be **suspended**, and it temperorily suspends its activity. A suspended thread can be **resumed**, and it is allowed to pick up from where it left off. A thread can be **blocked** when waiting for a resource. At any time, a thread can be **terminated**, which halts its execution immediately. Once terminated a thread cannot be resumed.

Thread priorities : Java assigns to each thread a priority. The priority determines how that thread should be treated with respect to others. Thread priorities are integers which specify the relative priority of one thread to another. The priority of a thread is used to decide when to switch from one running thread to the next. This is called as *context switch*.

Messaging : Once a program has been divided into separate threads, it is necessary to define how they will communicate with each other. Java provides a way for two or more threads to communicate with each other via call to predefined methods that all objects have.

7.1 Check Your Progress.

1. Fill in the blanks.

a) Multithreading is a specialised form of ....

b) The .... determines how a thread should betreated with respect to others.

c) .... are heavyweight, whereas .... are lightweight.

2. Write True or False.

a) Java provides built in support for multithreaded programming.

b) It is not possibleto temperorily suspend a thread which is running.

c) A thread once terminated can be resumed.

7.2 THE THREAD CLASS

Java's multithreading system is built upon the **Thread** class, its methods and the **Runnable** interface. **Thread** encapsulates a thread of execution. To create a new thread, your program should either extend the **Thread** or implement the **Runnable** interface.

The Main Thread : When a Java program starts up, one thread begins running immediately. This thread is usually called the main thread of the program since it is the one that is executed when the program begins. The main thread has the following features :

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

The main thread is automatically created when the program starts. It can be controlled through a **Thread** object. You have to obtain a reference to it by calling the method **currentThread()** of the **Thread** class. Its general form is :

```
static Thread currentThread()
```

This method is **static** and returns a reference to the thread in which it is called. With this you can obtain a reference to the main thread and can then control it like any other thread.

The following example illustrates :

```
class ThreadDemo {
                public static void main(String args[ ]) {
                        Thread th = Thread.currentThread();
                        System.out.println("Current thread is : " + th);
                        th.setName("MyThread");
                        System.out.println("After    chaging    name    to
MyThread : " + th);
                }
}
```

The output of the program is :

```
Current Thread : Thread[main, 5, main]
After Changing name to MyThread : Thread[MyThread, 5, main]
```

Here we obtain a reference to the current thread with the **currentThread()** method and store this reference in a **Thread** variable th. With the **println()** statement you display the information about the thread. As you can see in the output, by default the name of the thread is main, its priority is 5 and main is also the name of the group to which this thread belongs. A thread group is a data structure that controls the state of a collection of threads as a whole. The **setName()** method can be used to change the internal name of the thread. Then again th is output. The new name of the thread is now displayed.

sleep() is a method defined by **Thread** which causes the thread from which it is called to suspend execution for the specified period of milliseconds. Its general form is :

static void sleep(long *milliseconds*) throws InterruptedException

milliseconds specifies the number of milliseconds to suspend the thread. This method may throw an **InterruptedException**.

With the **setName()** method you can set the name of a thread. Similarly by using the **getName()** method you can obtain the name of a thread. These methods are :

final void setName(String *threadName*)

final String getName()

threadName specifies the name of the thread.

7.2 Check Your Progress.

1. Match the following :

	Column A		**Column B**
a)	main thread	i)	used to obtain name of a thread
b)	currentThread()	ii)	used to set name of a thread
c)	sleep()	iii)	used to obtain reference to current thread
d)	setName()	iv)	last thread to finish execution
e)	getName()	v)	causes the thread from which it is called to suspend execution

7.3 CREATING A THREAD

You can create a thread by creating an instance of type **Thread**. This can be done in two ways :

- Implement the **Runnable** interface
- Extend the **Thread** class itself.

7.3.1 Implementing Runnable

You create a class that implements **Runnable** interface to create a thread. You can construct a thread on any object that implements **Runnable**. In order to implement **Runnable**, the class needs to implement only one method **run()**. Its form is :

```
public void run()
```

Inside **run()**, you define the code which constitutes the new thread. Remember that **run()** can call other methods, use other classes and also declare variables just like the main thread can. The only difference is that **run()** establishes the entry point for another concurrent thread of execution within your program. This thread ends when **run()** returns.

Once you create a class that implements **Runnable**, you instantiate an object of type **Thread** from within the class. The constructor for **Thread** which we shall use is :

Thread(Runnable *threadObj*, String *threadName*)

threadObj is an instance of a class that implements the **Runnable** interface and this defines where the execution of the thread will begin. *threadName* is used to specify the name of the new thread.

A new thread does not start running, until you call its **start()** method. This method is declared within **Thread**. **start()** executes a call to **run()** and has the following form :

void start()

The following example illustrates how to create a new thread and the use of the **start()** method :

```
class NewThread implements Runnable {
                        Thread t ;
                        NewThread() {
                              t = new Thread(this, "Demo Thread");
                              System.out.println("Child Thread : " + t);
                              t.start();
                        }
                        public void run() {
                              try {
                                      for(i = 5; i > 0; i--) {
                                              System.out.println("Child
                                      Thread : " + i);
                                      Thread.sleep(500);
                        }
} catch(InterruptedException e) {

System.out.println("Interrupted
Child Thread");
                              }
                              System.out.println("Exiting Child Thread");
                        }
}
class ThreadDemo {
      public static void main(String args[ ]) {
                              new NewThread();
                              try {
```

```
                    for(int i = 5; i > 0 ; i--) {
                              System.out.println("In
                    Main Thread : " +i);
                              Thread.sleep(1000);
                    }
          } catch(InterruptedException e) {
                    System.out.println("Main Thread
          Interrupted");
          }
          System.out.println("Exiting Main Thread");
     }
}
```

In this program we create a **Thread** object with the following statement :

t = new Thread(this, "Demo Thread");

Passing this as the first argument indicates that you want the new thread to call the **run()** method on this object. **start()** method starts the execution of thread which begins at the **run()** method. The child thread's **for** loop begins execution. After calling **start()**, NewThread's constructor returns to **main()**. When the main thread resumes, it enters its **for** loop. Both the threads continue running, sharing the CPU till their respective loops finish.

The output of the program is :

```
Child thread : Thread[Demo Thread, 5, main]
In Main Thread : 5
Child Thread : 5
Child Thread : 4
In Main Thread : 4
Child Thread : 3
Child Thread : 2
In Main Thread : 3
Child Thread : 1
Exiting Child Thread
In Main Thread : 2
In Main Thread : 1
Exiting Main Thread
```

Remember that in a multithreaded program, the main thread must be the last thread to finish running. In case the main thread finishes before a child thread has completed, then the Java run time system may hang. Therefore we make the main thread sleep for 1000 milliseconds between iterations and the child thread for 500 milliseconds. This causes the child thread to terminate first.

7.3.2 Extending Thread

In the second method, you create a new class that extends **Thread** and then create an instance of that class. The extending class has to override the **run()** method which is the entry point for the new thread. It must also call the **start()** method to begin execution of the new thread. Let us rewrite the above program by extending **Thread** :

```
class NewThread extends Thread {
        NewThread() {
            super(“Demo Thread”);
            System.out.println(“Child Thread : “ + this);
            start();
        }
        public void run() {
            try {
                for(int i = 5; i > 0; i--) {
                    System.out.println(“In
                Child Thread : “ + i);
                    Thread.sleep(500);
                }
                } catch(InterruptedException e) {
                    System.out.println(“Child
                Interrupted”);
                }
            }
            System.out.println(“Exiting  child thread”);
        }
}
class ExtendThread {
        public static void main(String args[ ]) {
            new NewThread();
            try {
                for(int i = 5; i > 0 ; i--) {
                    System.out.println(“In
                Main Thread : “ +i);
                    Thread.sleep(1000);
                }
                } catch(InterruptedException e) {
                        System.out.println(“Main Thread
                Interrupted”);
                }
                System.out.println(“Exiting Main Thread”);
        }
}
```

This program generates the same output as the previous one. We make use of **super()** inside NewThread.

7.3 Check Your Progress.

1. Answer the following.

a) List the methods which can be used to create a thread by creating an instance of type **Thread**.

____

____

b) Which is the method used to start running a thread? What is its form?

____

____

7.4 CREATING MULTIPLE THREADS

In the following program we shall see how to create multiple child threads.

```
class NewThread implements Runnable {
        String name;
        Thread t;
        NewThread(String threadname) {
                name = threadname;
                t = new Thread(this, name);
                System.out.println(“New thread : “ + t);
                t.start();
        }
        public void run() {
                try {
                        for(int i = 5; i > 0; i--) {
                                System.out.println(name + “:” +
                        i);
                                Thread.sleep(1000);
                        }
                } catch(InterruptedException e) {
                                System.out.println(name +
                        "interrupted");
                        }
                System.out.println("Exiting" + name +
        "thread");
        }
}
class MultipleThread {
        public static void main(String args[ ]) {
                new NewThread("First");
                new NewThread("Second");
                new NewThread("Third");
                try {
                        Thread.sleep(10000);
                } catch(InterruptedException e) {
                        System.out.println("Main thread
                interrupted");
                }
                System.out.println("Exiting main thread");
        }
}
```

Run the program to determine what output it produces. Once started all the three child threads share the same CPU. Note that we have used the **sleep()** method in **main()** where we put the main thread to sleep for 10000 milliseconds. This ensures that all the child threads finish before main thread which should finish last.

7.5 ISALIVE() AND JOIN()

In all the preceding examples we called the **sleep()** method in **main()** in order to ensure that the main thread finishes last. However the **Thread** class defines a method to determine whether a thread has finished.

This method is **isAlive()** and its general form is :

final boolean isAlive()

If the thread upon which the method is called is still the running the method returns **true** else it returns a **false**.

One more method which is commonly used to wait for a thread to finish is **join()**. Its form is :

final void join() throws InterruptedException

This method waits until the thread on which it is called terminates. Other forms of **join()** are also there where you can specify the maximum amount of time that you want to wait for the specified thread to terminate.

Let us modify the previous program to make use of the **join()** method and ensure that main thread finishes last.

```
class NewThread implements Runnable {
        String name;
        Thread t;
        NewThread(String threadname) {
                name = threadname;
                t = new Thread(this, name);
                System.out.println(“New thread name : “ + t);
                t.start();
        }
        public void run() {
                try {
                        for( int i = 3; i > 0; i --) {
                                System.out.println(name + “:” +
                        i);
                                Thread.sleep(1000);
                        }
                } catch(InterruptedException e) {
                        System.out.println(“Thread “ + name +
                “interrupted”);
                        }
                System.out.println(“Exiting “ + name);
        }
}
class JoinDemo {
        public static void main(String args[ ]) {
                NewThread t1 = new NewThread(“First”);
                NewThread t2 = new NewThread(“Second”);
                NewThread t3 = new NewThread(“Third”);
                System.out.println(“First thread is alive” +
        t1.t.isAlive());
```

```
                              System.out.println("Second thread is alive"
+ t2.t.isAlive());
                              System.out.println("Third thread is alive" +
                        t3.t.isAlive());
                              try {
                                      System.out.println("Using join to
                              wait for threads to finish");
                                      t1.t.join();
                                      t2.t.join();
                                      t3.t.join();
                              } catch(InterruptedException e) {
                                      System.out.println("Main thread
                              interrupted");
                              }
                              System.out.println("First thread is alive " +
                        t1.t.isAlive());
                              System.out.println("Second thread is alive "
+ t2.t.isAlive());
                              System.out.println("Third thread is alive " +
t3.t.isAlive());
                        }
}
```

Sample output of the program is :

```
New thread : Thread[First, 5, main]
New thread : Thread[Second, 5, main]
New thread : Thread[Third, 5, main]
First thread is alive : true
Second thread is alive : true
Third thread is alive : true
First : 3
Second : 3
Third : 3
First : 2
Second : 2
Third : 2
First : 1
Second : 1
Third : 1
```

```
Exiting First
Exiting Third
Exiting Second
First thread is alive : false
Second thread is alive : false
Third thread is alive : false
```

7.4 & 7.5 Check Your Progress.

1. Fill in the blanks.

a) The .... thread should finish last when multiple threads have been created.

b) Thread class defines the .... method to determine whether a thread has finished.

c) The method which is commonly used to wait for a thread to finish is ....

7.6 THREAD PRIORITIES

Java assigns to each thread a priority. The priority determines how that thread should be treated with respect to others. Thread priorities are integers which specify the relative priority of one thread to another. The priority of a thread is used to decide when to switch from one running thread to the next. This is called as **context switch**. Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. Theorotically higher priority threads get more CPU time than the threads with lower priority. However in practice, the amount of CPU time that a thread gets depends upon a number of factors. The **setPriority()** method, which is a member of the **Thread** class is used to set the priority of a thread. Its general form is :

```
final void setPriority(int level)
```

where level specifies the new priority setting for the calling thread. The value of level should be within the range **MIN_PRIORITY** and **MAX_PRIORITY**. These values are 1 and 10 respectively. To return a thread to its default priority you use specify **NORM_PRIORITY** which is currently 5. These priorities are defined as final values in the **Thread** class.

The current priority can be obtained using the **getPriority()** method. Its form is :

```
final int getPriority()
```

It is important to note that implementations of Java have radically different behaviour when it comes to scheduling. Some versions like Windows works as you would expect, other versions may work quite differently.

Let us write a program to set two threads to different priorities. The instance hi is set to a higher priortiy than **NORM_PRIORITY** and lo is set to a priority less than the **NORM_PRIORITY**. Both the threads will increment the values of their counter within the alloted CPU time.

```
class Demo implements Runnable {
                    int i = 0;
                    Thread t;
                    private volatile boolean running = true;
                    public Demo(int p) {
                        t = new Thread(this);
                        t.setPriority(p);
                    }
                    public void run() {
                        while(running)
                              i++;
                    }
                    public void stop() {
                        running = false;
                    }
                    public void start() {
                        t.start();
                    }
}
class TPriority {
                    public static void main(String args[ ]) {

Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
                        Demo hi = new
                    Demo(Thread.NORM_PRIORITY + 2);
                        Demo lo = new
                    Demo(Thread.NORM_PRIORITY - 2);
                        lo.start();
                        hi.start();
                        try {
                              Thread.sleep(10000);
                        } catch(InterruptedException e) {
                              System.out.println("Main thread
                        interrupted");
                        }
                    lo.stop();
                    hi.stop();
                    try {
                            hi.t.join();
                            lo.t.join();
                    } catch(InterruptedException e) {
                    System.out.println("InterruptedException
            caught");
                    }
                    System.out.println("Low priority Thread : " +
      lo.i);
                    System.out.println("High priority Thread :" +
      hi.i);
            }
      }
```

The output of this program will vary from machine to machine and will also depend upon a number of factors like the CPU speed of the machine on which it is running and the other tasks that may be running on the system. In this example, running is preceded by the keyword **volatile**. **volatile** ensures that the value of running is examined each the the loop iterates. If the **volatile** keyword is not used, Java is free to optimise the loop as follows : It stores the value of running in a register of the CPU and does not necessarily check it at every iteration.

7.6 Check Your Progress.

1. Write True or False.

a) A priority is assigned to each thread by Java.

b) Thread priorities are integer values.

c) It is not possible to obtain the current priority of a thread.

d) All implementations of Java have similar behaviour when it comes to scheduling.

7.7 SYNCHRONIZATION

It is possible that two or more threads are required to access a common resource. Therefore there has to be some way to ensure that the resource would be used by only one thread at a time. The process by which this is achieved is called as synchronization. A monitor is an object that is used as a mutually exclusive lock or mutex. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All the other threads which are attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These threads are said to be waiting for the monitor. It is also possible for a thread which owns a monitor to re-enter the same monitor.

In order to synchronize the code, Java provides two ways, both of which make use of the **synchronized** keyword. Let us study these methods.

7.7.1 Synchronized methods

In Java, all objects have their own implicit monitor associated with them. In order to enter an object's monitor, you just call a method which has been modified by the keyword **synchronized**.

When a thread is inside a synchronized method, all the other threads which try to call that method or any other synchronized method, on the same instance have to wait. When the owner of the monitor returns from the synchronized method, it exits the monitor and relinquishes the control of the object to the next waiting thread.

In the following example, we shall create a synchronized method in a class. This program has three classes. The first class called First has a method **display()** which outputs a string. After the method is called, we put the thread to sleep for one second. The second class, Second has a constructor which takes a reference to an instance of First and a String. This constructor creates a new thread which will call the run method of this object. The **run()** method of Second, calls the display() method on the instance of First by passing the message. The last class is the SynchDemo which creates an instance of First, and three instances of Second and passes a String message. However, the same instance of First is passed to Second.

In this situation, if you do not synchroize the display() method, it allows the execution to switch to another thread by calling **sleep()**. Therefore, there is nothing to stop all the three threads from calling the same method, on the same object at the same time. This is called as *race condition*, because the three threads are racing each other to complete the method. To overcome this, you have to serialize the access to display(). This implies that you restrict the access to only one thread at a time and therefore you precede the display() method with the **synchronized** keyword.

```
class First {
        synchronized void display(String s) {
            System.out.println(s);
            try {
                Thread.sleep(1000);
            } catch(InterruptedException e) {
                System.out.println("Interrupted");
            }
            System.out.println("***");
        }
}
```

```
class Second implements Runnable {
        String s;
        First f;
        Thread t;
        public Second(First f1, String s1) {
                f = f1;
                s = s1;
                t = new Thread(this);
                f.start();
        }
        public void run() {
                t.display(s);
        }
}
class SyncDemo {
        public static void main(String args[ ]) {
                First f = new First();
                Second ob1 = new Second(f, “First”);
                Second ob2 = new Second(f, “Second”);
                Second ob3 = new Second(f, “Third”);
                try {
                        ob1.t.join();
                        ob2.t.join();
                        ob3.t.join();
                } catch(InterruptedException e) {
                        System.out.println(“Interrupted”);
                }
        }
}
```

As an exercise, first run this program without the use of the **synchronized** keyword with the display() method. You will obtain a mixed up output of the three message strings. Then run the same program by using the **synchronized** keyword.

Thus any time you have a method or a group of methods which manipulates the internal state of an object in a multithreaded situation, you have to use the **synchronized** keyword to prevent race conditions. Once a thread enters a synchronized method on an instance, no other thread can enter any other synchronized method on that instance. But remember, non synchronized methods on that instance can still be called.

7.7.2 synchronized statement

There may be situations where you may want to synchronize access to objects of a class which has not been designed for multithreaded access i.e. the class does not have any synchronized methods. Also if a class has been created by a third party where you have no access to the source code, it is not possible for you to precede the methods of that class by the **synchronized** keyword. For this purpose, you put calls to the methods defined by this class inside a synchronized block. The generalised form of the synchronized statement is :

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

where object is a reference to the object being synchronized. A synchronized block ensures that a call to a method which is a member of object will occur only after the current thread has successfully entered the object’s monitor.

Let us modify the previous program itself, where the display() method is not preceded by the keyword **synchronized**, but the synchronized statement is used in the **run()** method of Second. This program will produce the same output. Each thread will wait for the previous thread to finish before proceeding.

```
class First {
                    void display(String s) {
                    System.out.println(s);
                    try {
                         Thread.sleep(1000);
                    } catch(InterruptedException e) {
                         System.out.println(“Interrupted”);
                    }
                    System.out.println(“***”);
                }
}
class Second implements Runnable {
                String s;
                First f;
                Thread t;
                public Second(First f1, String s1) {
                    f = f1;
                    s = s1;
                    t = new Thread(this);
                t.start();
        }
        public void run() {
                synchronized(f) {
                f.display(s);
                }
        }
}
class SyncDemo {
        public static void main(String args[ ]) {
                First f = new First();
                Second ob1 = new Second(f, “First”);
                Second ob2 = new Second(f, “Second”);
                Second ob3 = new Second(f, “Third”);
                try {
                        ob1.t.join();
                        ob2.t.join();
                        ob3.t.join();
                } catch(InterruptedException e) {
                        System.out.println(“Interrupted”);
                }
        }
}
```

7.7 Check Your Progress.

1. Answer the following.

a) What is synchronization?

____

____

b) What does a synchronized block ensure?

____

____

7.8 INTERTHREAD COMMUNICATION

Java includes interprocess communication mechanism via the **wait()**, **notify()** and **notifyAll()** methods. All these methods are implemented as final methods in **Object** so all classes have them. All these methods can be called only from within a **synchronized** method. These methods are declared in **Object** as :

final void wait() throws InterruptedException

final void notify()

final void notifyAll()

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

notify() wakes up the first thread that called **wait()** on the same object.

notifyAll() wakes up all the threads that called **wait()** on the same object. The thread with the highest priority will run first.

These methods have just been introduced to the student. A detailed study of interthread communiction via these methods is not included in the study material.

7.9 DEADLOCK

A deadlock situation occurs when two threads have a circular dependancy on a pair of synchronized objects. eg. Suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y, it will block. However, if the thread in Y tries to call any synchronized method on X, the thread waits forever, because to access X it will have to release its own lock on Y so that the first thread could complete.

7.10 SUSPENDING, RESUMING AND STOPPING THREADS

Earlier versions of Java, had the methods **suspend()**, **resume()** and **stop()** defined by **Thread** to control a thread. **suspend()** and **resume()** are methods which pause and restart the execution of a thread. However these methods have been deprecated by Java 2. Therefore, a thread should be designed in such a manner that **run()** periodically checks to determine whether that thread should suspend, resume or stop its own execution. To accomplish this normally a flag variable is established that indicates the execution state of the thread. As long as this flag is set to running, the **run()** method should continue to let the thread execute, if this variable is set to suspend, the thread must pause and if it is set to stop, the thread must terminate.

7.8 to 7.10 Check Your Progress.

1. **Answer the following.**

a) What is the use of the wait() method?

____

____

b) When does a deadlock situation occur?

____

____

c) How should a thread be designed in versions of Java, where the uspend(), resume() and stop methods have been deprecated?

____

____

7.11 SUMMARY

A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.

Thread priority determines how that thread should be treated with respect to others. Once a program has been divided into separate threads, it is necessary to define how they will communicate with each other. Java provides a way for two or more threads to communicate with each other via call to predefined methods that all objects have.

Synchronization: It is possible that two or more threads are required to access a common resource. Therefore there has to be some way to ensure that the resource would be used by only one thread at a time. The process by which this is achieved is called as synchronization.

Source : *r4r.co.in* *(Link)*

7.12 CHECK YOUR PROGRESS - ANSWERS

7.1

1. a) multitasking

b) priority

c) Processes, threads

2. a) True

b) False

c) False

7.2

1. a) iv)

b) iii)

c) v)

d) ii)

e) i)

7.3

1. a) The ways used to create a thread by creating an instance of type **Thread** are : implement the **Runnable** interface and Extend the **Thread** class itself.

b) The **start()** method declared within **Thread** executes a call to **run()** and is used to make a thread run. Its form is : void start().

7.4 & 7.5

1. a) main

b) isAlive()

c) join()

7.6

1. a) True

b) True

c) False

d) False

7.7

1. a) The process by which it is ensured that a common resource required by two or more threads would be used by only one thread at a time is called as synchronization.

b) A synchronized block ensures that a call to a methodwhich is a member of object will occur only after the current thread has successfully entered the object's monitor.

7.8 to 7.10

1. a) The use of the **wait()** method is that it tells the calling thread to give up the monitor and go to sleep until someother thread enters the same monitor and calls **notify()**.

b) A deadlock situation occurs when two threads have a circular dependancy on a pair of synchronized objects.

c) A thread should be designed in such a manner that **run()** periodically checks to determine whether that thread should suspend, resume or stop its own execution in versions of Java where the suspend(), resume() and top() methods have been deprecated.

7.13 QUESTIONS FOR SELF - STUDY

1. Compare multiprogramming and multithreading.
2. Write short notes on the following :
   a) States of a thread
   b) Implementing Runnable
   c) Extending Thread
   d) Thread Priorities
   e) Deadlock
3. What is the main thread? Describe its features.
4. Describe the methods : currentThread() and sleep() alongwith their general forms.
5. Describe in brief the use of isAlive() and join() and write the general forms of these methods.
6. Describe synchronization. Describe the ways in which code can be synchronized in Java.
7. List and describe briefly the methods used for interthread communication.

7.14 SUGGESTED READINGS

1. www.java.com
2. www.freejavaguide.com
3. www.java-made-easy.com
4. www.tutorialspoint.com
5. www.roseindia.net

❑ ❑ ❑

Notes

Notes

Chapter : 8

Strings

8.0 OBJECTIVES

Dear Students,

After studying this chapter you will able to :

- describe how to creat String objects and constructors of String
- discuss special String operations
- discuss the String comparison functions
- explain the character extraction functions
- describe the various functions to modify, compare and search strings.
- discuss what is meant by command line arguments.

8.1 INTRODUCTION

We have already studied that in Java a string is a sequence of characters.

Java implements strings as objects of type **String**. **String** objects can be constructed in a number of ways, therefore it is easy to obtain a string when needed. Java has a number of methods to perform various operations on strings like comparing strings, concatenating strings, changing case of letters within a string etc.

Remember that once a **String** object has been created, you cannot change the characters that comprise the string. This means that the contents of the **String** instance cannot be changed after it has been created. However, a variable declared as a **String** reference can be changed to point to some other **String** object anytime.

In those cases, that you require that a string is to be changed, there is an alternative class called **StringBuffer**, whose objects contain strings that can be modified after they are created. Both **String** and **StringBuffer** are defined in **java.lang**. This implies that they are automatically available to programs. Both are declared **final**, therefore they cannot be subclassed.

String represents fixed length, immutable character sequences. On the other hand, **StringBuffer** represents growable and writeable character sequences. **StringBuffer** may have characters and substrings inserted in the middle or appended at the end. **StringBuffer** will automatically grow to make room for such additions. We shall cover **String** and not address **StringBuffer** in this study material.

8.1 Check Your Progress.

1. Fill in the blanks.

a) Java implements strings as objects of type ....

b) The contents of the .... instance cannot be changed after it has been created.

c) .... is the class whose objects contain strings that can be modified after they are created.

8.2 THE STRING CONSTRUCTORS

Let us see the various constructors supported by the **String** class.

- To create an empty **String**, you call the default constructor eg.

  String s = new String();

  This will create an instance of **String** which has no characters.

- To create a **String** initialised by an array of characters we can use the following constructor :

  String(char *chars*[])

  eg. char chars[ ] = { 'a', 'b', 'c' };

  String s = new String(chars);

  This constructor will initialise s with the string "abc".

- A subrange of a character array can also be specified as an initialiser for a string. Its constructor is:

String(char *chars*[ ], int *startIndex*, int *numChars*)

startIndex specifies the index at which the subrange begins, and *numChars* specifies the number of characters to be used. eg.

```
char chars[ ] = {‘a’, ‘b,’ ‘c’, ‘d’, ‘e’ , ‘f’, ‘g’};
String s = new String(chars, 3,2);
```

This set of statements will initialise s with the characters cd

You can construct a **String** object that contains the same character sequence as another **String** object. You use the following constructor for this purpose :

String(String *strObj*)

where *strObj* is a **String** object.

The following example will illustrate :

```
class StringDemo {
    public static void main(String args[ ]) {
            char c[ ] = {‘a’, ‘b’, ‘c’, ‘d’, ‘e’};
            String s1 = new String(c);
            String s2 = new String(s1);
            System.out.println(s1);
            System.out.println(s2);
    }
}
```

The output of this program will be :

```
abcde
abcde
```

This shows that s1 and s2 both contain the same string.

We have already studied that **char** uses 16 bits to represent the Unicode character set. A typical format on the Internet however uses arrays of 8 bit bytes constructed from the ASCII character set. For this purpose the **String** class provides constructors which initialise a string when given a byte array. Their forms are :

String(byte *asciiChars*[ ])

String(byte *asciiChars*[ ], int *startIndex*, int *numChars*])

where *asciiChars* specifies the array of bytes. As you can see, the second form will allow you to specify a subrange. In both these constructors, the byte to character conversion is done by using the default character encoding of the platform.

The following example will illustrate the use of these constructors :

```
class Strings {
        public static void main(String args[ ])      {
                byte a[ ] = {65, 66, 67, 68, 69, 70, };
                String s1 = new String(a);
                System.out.println(s1);
                String s2 = new String(a, 2, 3);
                System.out.println(s2);
        }
}
```

The output of the program will be:

```
ABCDE
CDE
```

String length :

The length of the string is the number of characters it contains. The **length()** method is used to determine the length of the string. Its form is :

int length()

Thus it returns the value of type **int**.

eg. int i = s.length();

where s is an object of type **String**.

8.2 Check Your Progress.

1. Write the String constructors for the following.

a) To create an empty String : ....

b) To create a String initialised by an array of characters : ....

c) To initialise a string when given a byte array : ....

8.3 SPECIAL STRING OPERATIONS

Let us study some special string operations in this section.

8.3.1 String Literals

In the previous examples we saw how to create an instance of **String** from an array of characters with the use of the **new** operator. However, there is an easier and more convenient way to do this. You can use a string literal to initialise a **String** object. For each literal in your program, Java automatically constructs a **String** object.

eg. String s1 = “abc”;

will create s1 with contents abc.

A **String** object is created for every string literal, therefore you can use a string literal at any place where you can use a String object. eg. you can call methods directly on a quoted string as if it were an object reference. eg.

```
System.out.println("abc".length());
```

8.3.2 String Concatenation

Java does not allow operators to be applied to **String** objects. The only exception to this rule is the + operator, which concatenates two strings and produces a **String** object as the result. eg.

```
String s1 = "Good";
String s2 = "Morning";
String s3 = s1 + s2 + "to you";
System.out.println(s3);
```

will output “Good Morning to you”

You can also concatenate strings with other data types. eg.

```
int time = 8;
String s1 = "It is " + time + "o'clock";
System.out.println(s1);
```

In this case, time is an **int** data type. The **int** value in time will automatically get converted into its string representation within a **String** object. The string will then be concatenated and the output will be:

```
It is 8 o'clock
```

8.3.3 toString()

When Java converts data into its string representation during concatenation, it does so by calling one of the overloaded versions of the string conversion method **valueOf()** defined by **String**. **valueOf()** is overloaded for all the simple types and for type **Object**. For the simple types **valueOf()** returns a string that contains the human readable equivalent of the value with which it is called. For objects, **valueOf()** calls the **toString()** method on the object.

Every class implements **toString()** because it is defined by **Object**. Many times you will want to override **toString()** and provide your own string representations. This is easily accomplished. The general form of **toString()** is :

String toString()

Thus just returning a **String** object that contains the human readable string which describes an object of your class is sufficient.

By overriding **toString()** for the classes that you create, you allow the resulting strings to be fully integrated into Java’s programming environment. Thus they can be used in the **print()** and **println()** statements and in concatenation. The following example illustrates :

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

```
                        i = a;
                        j = b;
                }
                public String toString() {
                        return "Value of i " + i + "and value of j " + j;
                }
        }
        class DemotoString {
                public static void main(String args[]) {
                        StringsDemo s = new StringsDemo(10,
        100);
                        System.out.println(s);
                        String msg = "Automatic invokation of
        String method : " + s;
                        System.out.println(msg);
                }
        }
```

The output of this program is :

```
Value of i 10 and value of j 100
Automatic invokation of String method : Value of i 10 and value of j 100
```

Thus you can see that the method **toString()** is invoked automatically whenever you use an instance of StringsDemo in **println()**.

8.3 Check Your Progress.

1. Write True or False.

a) A string literal cannot be used to initialise a String object.

b) The + operator can be applied to String objects.

c) Every class implements toString().

8.4 EXTRACTING CHARACTERS

In this section let us see the different ways in which characters can be extracted from a **String** object.

8.4.1 charAt()

With this method you can directly refer to a single character. Its general form is :

char charAt(int *where*)

where is the index of the character you want to obtain. Remember that the value of where must not be negative and should specify a location within the string.

eg. char ch;

ch = “Mystring”.charAt(4) will assign the value 'r' to ch.

8.4.2 getChars() :

The general form of **getChars()** is :

void getChars(int *sourceStart,* int *sourceEnd*, char *target*[ ], int *targetStart*)

sourceStart specifies the index of the beginning of the substring, and *sourceEnd* specifies the index that is one past the end of the desired substring. This means that the substring will contain characters from *sourceStart* to *sourceEnd*-1. The receiving array is specified by *target*. The index at which the substring will be copied into target is specified by *targetStart*. It should be ensured that target is large enough to hold the specified number of characters.

The following example will demonstrate use of **getChars()**

```
class StringsDemo {
      public static void main(String args[]) {
                  String s1 = “Extracting characters from
String”;
                  start = 4;
                  end = 10;
                  char substring[] = new char[end - start];
                  s1.getChars(start, end, substring, 0);
                  System.out.println(“The extracted
characters : “);
                  System.out.println(substring);
      }
}
```

The output of the program will be :

```
acting
```

8.4.3 getBytes() :

Is a variation of **getChars()**, in which the characters are stored in an array of bytes. This method uses the default character to byte conversions provided by the platform. Its simplest form is :

byte[ ] getBytes()

8.4.4 toCharArray() :

This method is used if you want to convert all the characters in a **String** object into a character array. The general form of this method is :

char[ ] toCharArray()

Note that this method returns an array of characters for the entire string.

8.4 Check Your Progress.

1. Match the following.

	Column A		**Column B**
a)	charAt()	i)	characters are stored in an array of bytes.
b)	getBytes()	ii)	Converts all the characters in a String object into a character array
c)	toCharArray()	iii)	can directly refer to a single character

8.5 COMPARING STRINGS

In this section let us study some method used to compare strings and substrings.

equals() : The general form of **equals()** is :

boolean equals(Object *str*)

str is the **String** object to be compared with the invoking **String** object. This method returns **true** if the strings are the same, i.e. they contain the same characters in the same order, else returns **false**.

equalsIgnoreCase() : With this method, a comparison which ignores the case can be performed Its general form is :

boolean equalsIgnoreCase(String *str*)

where *str* is the *String* object to be compared with the invoking *String* object. As in the case of **equals()**, it returns a **true** if the strings are the same, i.e. they contain the same characters in the same order, else returns **false**.

Let us see an illustration of the above methods :

```
class StringsDemo {
        public static void main(String args[ ]) {
                String s1 = “Good”;
                String s2 = “gOOd”;
                String s3 = “Good”;
                System.out.println(s1 + “ equals ”  + s3 + “---> “
+ s1.equals(s3));
                System.out.println(s1 + “ equals ”  + s2 + “-
--> “ + s1.equals(s2));
                System.out.println(s1+“
        equalsIgnoreCase ”  + s2 + “---> “ +
s1.equalsIgnoreCase(s3));
        }
}
```

The output of the program will be :

```
Good equals Good ---> True
```

Good equals gOOd ---> False

Good equalsIgnoreCase gOOd ---> True

regionMatches() : This method compares a specific region inside a string with another specific region in another string. Its general form is :

boolean regionMatches(int *startIndex*, String *str2*, int *str2StartIndex*, int *numChars*)

An overloaded version of this method allows you to ignore cases in the comparison. Its general form is :

boolean regionMatches(boolean *ignoreCase*, int *startIndex*, String *str2*, int *str2StartIndex*, int *numChars*)

In both these methods, *startIndex* specifies the index at which the region begins within the invoking **String** object. *str2* specifies the String being compared. The index at which the comparison should start within *str2* is specified by *str2StartIndex*. *numChars* gives the number of characters to be compared. In the overloaded method, if *ignoreCase* is **true**, the case of characters is ignored. Else case is considered.

startsWith() and **endsWith()** : The **startsWith()** method determines whether a given **String** begins with a specified string, and **endsWith()** method determines whether the String ends with the specified string. The general forms of these methods are :

boolean startsWith(String *str*)

boolean endWith(String *str*)

str is the **String** to be tested. The method return **true** if the string matches, else it returns **false**.

eg.	MyString.startsWith("My")	returns **true**
	MYString.endsWith("ing")	returns **true**

It is also possible to provide a starting point in the **startWith()** method as follows:

boolean startsWith(String *str*, int *startIndex*)

where *startIndex* is the index at which point the search should begin. eg.

MyString.startsWith("Str", 2)   returns **true**

compareTo() : In many applications you need to know, whether a string is less than, equal to or greater than another string. A string is less than another string, if it comes before the other in dictionary order. A string is greater than the other string if it comes after the other in dictionary order. For this comparison the method is **compareTo()**. Its general form is :

int compareTo(String *str*)

where *str* is the **String** being compared with the invoking **String**. This method returns an **int** value, and the result is interpreted as follows :

Value	**Meaning**
Less than zero T	he invoking string is less than str
Greater than zero	The invoking string is greater than str
Zero	The two strings are equal.

The version of the method, to be used if you wish to ignore case is :

int compareToIgnoreCase(String *str*)

8.5 Check Your Progress.

1. Write True or False.

a) In Java, it is possible to perform a comparison which ignores the case.

b) A string is less than another string, if it comes before the other in dictionary order.

c) It is possible to determine whether a given **String** begins with a specified string.

8.6 SEARCHING STRINGS

You can search for a specified character or a substring within a given string with the following methods :

indexOf() - searches for the first occurance of a character or a substring

lastIndexOf() - searches for the last occurance of a character or a substring

The methods return the index at which the character or substring is found, else return a -1 if no match is found.

The general forms of the methods are :

int indexOf(int *ch*) - to search for the first occurance of a character

int lastIndexOf(int *ch*) - to search for the last occurance of a character

where ch is the character to be searched.

eg. tooth.indexOf('t') will return 0

tooth.lastIndexOf('t') with return 3

To search for the first or last occurance of a substring we use the overloaded form of the method as:

int indexOf(String *str*)

int lastindexOf(String *str*)

where *str* specifies the substring

For both these forms, it is possible to specify the starting point of the search using the following forms:

int indexOf(int *ch*,int *startIndex*)

int lastIndexOf(int *ch*, int *startIndex*)

int indexOf(String *str*, int *startIndex*)

int lastIndexOf(String *str*, int *startIndex*)

where *startIndex* specifies the index at which point the search begins. Remember that for **indexOf()** the search runs from *startIndex* to end of string, whereas for **lastIndexOf()** the search runs from *startIndex* to zero.

8.7 MODIFYING A STRING

Whenever you want to modify a string, use one of the following string methods.

8.7.1 substring

Using **substring()** you can extract a substring from a string. It has two forms. The first form is :

String substring(int *startIndex*)

here *startIndex* specifies the index at which the substring will begin. This form returns a copy of the substring which begins at *startIndex* and ends at the end of the invoking string.

In the second form, you can specify both the starting index and ending index of the substring :

String substring(int *startIndex*, int *endIndex*)

where *startIndex* specifies the beginning index and *endIndex* specifies the stopping point. The string returned will contain all characters from the beginning index, upto but not including the ending index.

eg. String s1 = “Mystring”.substring(2) will assign the substring "string" to s1.
String s = “Mystring”.substring(2,5) will assing "str" to s

8.7.2 concat

This method is used to concatenate two strings and has the following form :

String concat(String *str*)

This method creates a new object, that contains the invoking string and the contents of *str* appended to it at the end.

eg. String s2 = “New”.concat(“Strings”) will put "NewStrings" into s2.
concat() performs the same function as +.

8.7.3 replace()

This method replaces all occurances of one character in the invoking string with the specified character. Its general form is :

String replace(char *original*, char *replacement*)

The method returns the string, where the characater specified by *original* is replaced by the character specified by *replacement*.

```
eg. String s = “strings”.replace(‘s’, ‘a’);
```

will put "atringa" into s.

8.7.4 trim()

This method returns a copy of the invoking string from which any leading and trailing white spaces have been removed. Its general form is :

```
String trim()
eg. String s1 = “      Good  Morning    “.trim();
```

will put “Good Morning” into s1.

This method is very useful when you process user commands, where the leading and trailing spaces which the user may have inadvertently entered can be trimmed.

8.7.5 Changing Case

toLowerCase() converts all the characters in the string from uppercase to lowercase. The method **toUpperCase()** converts all the characters in a string from lowercase to uppercase. The general forms of these methods are :

```
String toLowerCase()
String toUpperCase()
```

The following program demonstrates the use of these methods :

```
class ChangeCase {
        public static void main(String args[ ]) {
                String s = “Convert case of This String”;
                System.out.println(“Original String : “ + s);
                String upper = s.toUpperCase();
                String lower = s.toLowerCase();
                System.out.println(“Upper Case String : “ +
upper);
                System.out.println(“Lower Case String : “ +
lower);
        }
}
```

The output of this program will be :

```
Original String : Convert case of This String
Upper Case String : CONVERT CASE OF THIS STRING
Lower Case String : convert case of this string
```

8.8 USING COMMAND LINE ARGUMENTS

In many situations it is likely that you will want to pass information to a program when you run it. This is accomplished by passing command line arguments to **main()**. A command line argument is the information which directly follows the name of the program on the command line, when the program is executed. In Java, the command line arguments are stored as strings in the **String** array passed to **main()**. Therefore they can be easily accessed as shown in the following example :

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

If you run this program at the command line as follows :

```
java CommandDemo Passing Command line arguments 1
```

then the output will be :

```
args[0] : Passing
args[1] : Command
args[2] : line
args[3] : arguments
args[4] : 1
```

Remember, command line arguments are passed as strings, so you should convert numeric values to their internal forms manually.

8.6 to 8.8 Check Your Progress.

1. Fill in the blanks.

a) .... searches for the first occurance of a character or a substring.

b) .... method returns a copy of the invoking string from which any leading and trailing white spaces have been removed.

c) .... converts all the characters in a string from lowercase to uppercase.

d) Command line arguments are passed as ....

8.9 SUMMARY

In Java a string is a sequence of characters. Java implements strings as objects of type String. String objects an be constructed in a number of ways, therefore it is easy to obtain a string when needed. Java has a number of methods to perform various operations on strings like comparing strings, concatenating strings, changing case of letters within a string etc.

Source : *train-srv.manipalu.com (Link)*

8.10 CHECK YOUR PROGRESS - ANSWERS

8.1

1. a) String

b) String

c) StringBuffer

8.2

1. a) String()

b) String(char chars[])

c) String(byte asciiChars[])

8.3

1. a) False

b) True

c) True

8.4

1. a) iii)

b) i)

c) ii)

8.5

1. a) True

b) True

c) True

8.6 to 8.8

1. a) indexOf()

b) trim()

c) toUpperCase()

d) strings

8.11 QUESTIONS FOR SELF - STUDY

1. Compare String and StringBuffer class.
2. Describe the various constructors of String.

3. Write short notes on :

a) String Literals

b) toString()

c) Command Line arguments.

4. Describe the various ways in which characters can be extracted from a String object.

5. Describe the various functions which are used to compare Strings in Java.

6. Describe the methods useful for searching Strings.

7. Which methods are useful for modifying Strings? Explain them.

8.12 SUGGESTED READINGS

1. www.java.com
2. www.freejavaguide.com
3. www.java-made-easy.com
4. www.tutorialspoint.com
5. www.roseindia.net

❑ ❑ ❑

Notes

Chapter : 9

Input/Output

9.0 OBJECTIVES

Dear Students,

After studying this chapter you will able to :

- discuss an overview of user interaction in Java
- discribe what are streams
- discuss methods for performing console input and console output
- discuss file input and output

9.1 INTRODUCTION

In Java, user interaction takes place through graphically oriented **applets** which depend upon the Abstract Window Toolkit. Thus most real applications of Java are not text based console programs. Also Java provides limited support to console Input/Output and it is not very important to Java programming. However, Java provides a strong flexible support for I/O as it relates to files and networks. In this chapter, let us familiarize ourselves with console and file input/output in Java.

9.2 STREAMS

I/O is performed through streams in Java. A stream is an abstraction which either produces or consumes information. A stream is linked to a physical device by the Java I/O system. The same I/O classes and methods can be applied to any type of device, because all the streams behave in the same manner irrespective of the physical devices to which they connect. An input stream can thus abstract various kinds of input like disk file, keyboard, network socket etc. Similarly an output stream may refer to the console, a disk file or a network connection. Java implements streams within class hierarchies defined in the **java.io** package.

Java defines two types of streams : **byte streams** and **character streams**. Byte streams provide a convenient way to handle input/output of bytes. They are useful when reading or writing binary data. Character streams provide a convenient way of handling input and output of characters. They use Unicode and can therefore be internationalised. In some cases, character streams are more efficient than byte streams. Note however, that at the lowest level all I/O is byte oriented. The character based I/O streams are just a convenient and efficient way to handle characters.

9.2.1 The Byte Stream Classes

Byte streams are defined by using two class hierarchies. At the top are two abstract classes : **InputStream** and **OutputStream**. Both these classes have a number of concrete subclasses which handle the differences between various devices like disks, network connections and memory buffers.

Some of the byte stream classes which we shall use in our examples are :

BufferedInputStream	Buffered Input stream
BufferedOutputStream	Buffered Output stream
DataInputStream	Input stream that contains methods for reading the Java standard data types
DataOutputStream	Output stream that contains mehtods for writing the Java standard data types

In order to use the stream classes you have to import **java.io**. There are abstract classes in **InputStream** and **OutputStream** which define several key methods which the other stream classes implement. The most important methods are **read()** which is used to read bytes of data and **write()** which is used to write bytes of data. Both the methods are declared **abstract**. They are overridden by the derived stream classes.

9.2.2 Character Stream Classes

They are also defined by using two class hierarchies. At the top are two abstract classes **Reader** and **Writer**. These classes handle the Unicode character streams. There are a number of concrete subclasses of each of these. Some of the character stream classes are :

The **Reader** and **Writer** classes define several methods which are implemented by the other stream classes. The most common among them are **read()** and **write()** which read and write characters of data, respectively.These methods are overridden by the derived stream classes.

Buffered Reader	Buffered Input Character Stream
Buffered Writer	Buffered output Character Stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file

9.2.3 Predefined Streams

We know that all Java programs automatically import the **java.lang** package. This package defines a class called **System**. **System** encapsulates several aspects of the Java run time environment. eg. with some of its method you can obtain the current time, the settings of various properties associated with the system etc. **System** contains three predefined stream variables **in**, **out** and **err** which are declared **public** and **static** within **System**. Thus they can be used by any other part of your program without specific reference to any specific **System** object.

System.out refers to the standard output stream, which by default is the console. **System.in** refers to the standard input which is by default the keyboard. **System.err** refers to the standard error stream, which is console by default. Of course, these streams may be redirected to any compatible I/O.

System.out and **System.err** are objects of type **PrintStream**. **System.in** is an object of type **InputStream**. Though they are used for reading and writing characters from and to the console. These are byte streams.

9.1 & 9.2 Check Your Progress.

1. Fill in the blanks.

a) .... depend upon the Abstract Window Toolkit.

b) I/O is performed through .... in Java.

c) At the lowest level all I/O is .... oriented.

d) In order to use the stream classes you have to import the....................... .

e) **System.out** refers to the standard output stream, which by default is the ....

2. Write True or False.

a) InputStream is an abstract class.

b) Reader class handles Unicode character streams.

c) System.err is object of type InputStream.

9.3 REDING CONSOLE INPUT

We saw in the preceding section that we make use of **System.in** for console input. To obtain a character based stream which is attached to the console, you wrap **System.in** in a **BufferedReader** object, to create a character stream. **BufferedReader** supports a buffered input stream. Its most commonly used constructor is :

BufferedReader(Reader *inputReader*)

where *inputReader* is the stream which is linked to the instance of BufferedReader that is being created. Remember **Reader** is an abstract class. One of its concrete classes is **InputStreamReader** which converts bytes to characters.

To get an **InputStreamReader** object which is linked to **System.in** we use the following constructor :

InputStreamReader(InputStream *inputStream*)

Now **System.in** refers to an object of type **InputStream**, so it can be used for *inputstream*. Thus we obtain the line of code to create a **BufferedReader** connected to the keyboard as follows:

```
    BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
```

Now br is a character based stream linked to the console through **System.in**.

9.3.1 Reading Characters

To read a character from a **BufferedReader** we use the **read()** method as :

int read() throws IOException

Every time a **read()** is called it reads a character from the input stream and returns it as an integer value. When the end of stream is encountered it returns -1. It can throw an **IOException**.

Let us write a program to read characters from the console :

```
import java.io.*;
class CharInput {
      public static void main(String args[]) throws IOException {
            char c;
            BufferedReader br =
                  new BufferedReader(new
            InputStreamReader(System.in));
            System.out.println("Enter character :");
            c = (char) br.read();
            System.out.println(c);
      }
}
```

A sample run of the program :

```
Enter character
a
a
```

Remember that **System.in** is by default line buffered. This means that no input will pass to the program till you press Enter. Thus you have to press Enter everytime you want **System.in** to read a character. **read()** is therefore not very valuable for interactive console input.

9.3.2 Reading Strings

readLine() is a method of the **BufferedReader** class which can be used to read a string from the keyboard. Its general form is :

```
String readLine() throws IOException
```

Note that it returns a **String** object. The following program demonstrates the use of **readLine()** to read and display lines of text till you enter the word Over.

```
import java.io.*;
class ReadLines {
        public static void main(String args[])
                throws IOException
        {
                BufferedReader br =
                        new BufferedReader(new
                InputStreamReader(System.in));
                String s1;
                System.out.println("Enter Strings and Over to
        quit");
                do {
                        s1 = br.readLine();
                        System.out.println(s1);
                } while(!s1.equals("Over"));
        }
}
```

9.3 Check Your Progress.

1. Fill in the blanks.

a) .... supports a buffered input stream.

b) To read a character from a BufferedReader we use the .... method.

c) Every time a **read()** is called it reads a character from the input stream and returns it as an .... value.

d) .... is a method of the **BufferedReader** class which can be used to read a string from the keyboard.

9.4 WRITING CONSOLE OUTPUT

We have used **print()** and **println()** to accomplish console output in all our previous examples. These methods are defined by the class **PrintStream** which is the type of the object referenced by **System.out**. Remember that **System.out** is a byte stream. **PrintStream** also implements the low level method **write()** since it is an output stream derived from **OutputStream**. Thus, **write()** can also be used to write to the console. The simplest form of **write()** is :

void write(int *byteval*) throws IOException

This method writes to the file the byte which is specified by *byteval*. Though byteval is declared as an integer, only the lower order eight bits are written. The following example will demonstrate use of **write()** :

```
class Write {
      public static void main(String args[]) {
                int i;
                i = ‘Z’;
                System.out.write(i);
                System.out.write(‘\n’);
      }
}
```

9.4.1 PrintWriter Class

The recommended method of writing to the console when using Java is through a **PrintWriter** stream. **PrintWriter** is one of the character based classes. Using a character based class for console input/output makes it easy to internationalise your programs. Therefore though **System.out** is permissible for console output, **PrintWriter** stream is what is recommended.

The constructor of **PrintWriter** which we shall use is :

PrintWriter(OutputStream *outputStream,* boolean *flushOnNewline*)

Here, *outputStream* is an object of type OutputStream and *flushOnNewline* controls whether Java flushes the output stream every time a ‘\n’ character is output. If *flushOnNewline* is **true**, flushing automatically takes place, and if **false**, then flushing is not automatic.

PrintWriter supports the **print()** and **println()** methods for all types including **Object**. Thus these methods can be used in the same way as they have been used with **System.out**. If an argument is not a simple type, then **PrintWriter** methods call the object’s **toString()** method and then print the result.

To write to the console using a **PrintWriter**, specify **System.out** for the output stream and flush the stream after each newline.

The following example demonstrates :

```
import java.io.*;
public class PrintWrite {
                public static void main(String args[]) {
                    PrintWriter pw = new
                PrintWriter(System.out, true);
                    pw.println(“A PrintWriter Demo string”);
                    int i = 10;
                    pw.println(i);
                    double d = 3.2e-4;
                    pw.println(d);
                }
}
```

The output of the program will be :

```
A PrintWriter Demo string
10
3.2E-4
```

9.4 Check Your Progress.

1. Write True or False.

a) The **println()** method is defined by the class **PrintStream**.

b) The **write()** method cannot be used to write to the console.

c) **PrintWriter** is a character based class.

d) **PrintWriter** cannot support the **println()** method for objects.

9.5 READING AND WRITING FILES

There are a number of classes provided by Java to read and write files. In Java, all files are byte oriented and Java provides the methods to read and write bytes from and to a file. We shall study the basics of file I/O with two of the most frequently used stream classes **FileInputStream** and **FileOutputStream**. These classes create bytestreams linked to files. To open a file you create an object of one of these classes by specifying the name of the file an an argument to the constructor. The forms of these classes which we shall be using are :

FileInputStream(String *fileName*) throws
FileNotFoundException

FileOuptutStream(String *fileName*) throws
FileNotFoundException

fileName specifies the name of the file to be opened. The **FileNotFound Exception** will be thrown if you create an input stream and the specified file does not exist. For output streams if the file cannot be created then the **FileNotFound** Exception will be thrown. If a file with the name already exists, then its contents will be destroyed.

You should close the file using **close()** once you have finished working with the file. This method is defined by both **FileInputStream** and **FileOutputStream** and is :

void close() throws IOException

Reading from a file :

In order to read from a file we use the following form of **read()** which is defined in the **FileInputStream**

int read() throws IOException

Every time **read()** is called it reads a single byte from the file and returns the byte as an integer value. When the end of file is encountered **read()** returns a value -1. Remember it can throw an **IOException**.

The following program uses **read()** to input and display contents of a text file :

```
import java.io.*;
class FileDemo {
        public static void main(String args[])
          throws IOException
        {
                int i;
                FileInputStream fin;
                try {
                        fin = new FileInputStream(args[0]);
                } catch(FileNotFoundException e) {
                System.out.println("File Not Found");
                return;
                } catch(ArrayIndexOutOfBoundsException
        e) {
                System.out.println("No filename specified");
                return;
                }
                do {
                        i = fin.read();
                        if(i != -1)
                                System.out.print((char) i);
                } while(i != -1);
                fin.close();
        }
```

}

Remember that in this program you will supply the filename whose contents you wish to display as a command line argument. Thus when you run this program you should type :

java FileDemo *filename*

where *filename* will be the name of the file which you wish to display. The name of the file will be passed as an argument to **FileInputStream** through args[0]. It will throw an exception if you do not give a name. Also remember that the program will throw an exception if the file is not found. So you should make use of the **try** statements as shown in order to check for these situations.

Writing to a file :

To write to a file, you make use of the **write()** method defined by the **FileOutputStream**.The simplest form of **write()** is :

void write(int byteval) throws IOException

This method writes the byte specified by byteval to the file. Although byteval is declared an integer, it will write only the low order eight bits. An **IOException** is thrown if an error occurs.

The following example copies contents of one file into another. This program should be run on the command line by specifying the name of the source file and the target file.

```
import java.io.*;
class CopyFile {
                    public static void main(String args[])
                     throws IOException
                    {
                         int i;
                         FileInputStream fin;
                         FileOutputStream fout;
                         try {
                                try {
                                       fin = new
                                FileInputStream(args[0]);
                                } catch(FileNotFoundException e) {
                                System.out.println("Source File not
                    found");
                                return;
                                }

                                try {
```

```
                    fout = new
FileOutputStream(args[1]);
                } catch(FileNotFoundException e) {
                System.out.println(“Error opening
        target file”);
                return;
                }
        } catch(ArrayIndexOutOfBoundsException
e) {
        System.out.println(“Error ! Incorrect
usage”);
        return;
        }
        try {
                do {
                i = fin.read();
                if(i != -1) fout.write(i);
                } while(i != -1);
        } catch(IOException e) {
        System.out.println(“File Error”);
        }
fin.close();
fout.close();
}
}
```

9.5 Check Your Progress.

1. Answer the following :

a) When will the **FileInputStream()** throw a **FileNotFoundException**?

____

____

b) Write the simplest form of the method **write()** when used to write to a file.

____

____

9.6 Summary

I/O is performed through streams in Java. A stream is an abstraction which either produces or consumes information. A stream is linked to a physical device by the Java I/O system.

Java defines two types of streams : byte streams and character streams. Byte streams provide a convenient way to handle input/output of bytes. There are useful when reading or writing binary data. Character streams provide a convenient way of handling input and output of characters. They use Unicode and can therefore be internationalised. At the lowest level all I/O is byte oriented.

9.7 CHECK YOUR PROGRESS - ANSWERS

9.1 & 9.2

1. a) applets

b) streams

c) byte

d) java.io

e) console

2. a) True

b) True

c) False

9.3

1. a) BufferedReader

b) read()

c) integer

d) readLine()

9.4

1. a) True

b) False

c) True

d) False

9.5

1. a) The **FileNotFoundException** will be thrown by the **FileInputStream()**, if you create an input stream and the specified file does not exist.

b) The simplest form of **write()** is :void write(int byteval) throws IOException

This method writes the byte specified by byteval to the file. Although byteval is declared an integer, it will write only the low order eight bits. An **IOException** is thrownif an error occurs.

9.8 QUESTIONS FOR SELF - STUDY

1. **Write Short notes on :**
   a) Byte Stream Classes
   b) Character Stream Classes
   c) Predefined Streams
   d) **PrintWriter** Class
2. Describe how to obtain a character based stream linked to the console through **System.in**.
3. Explain the methods used to read characters and strings from the console using **BufferedReader**.
4. Explain the **FileInputStream()** and **FileOutputStream()** in brief.

9.9 SUGGESTED READINGS

1. www.java.com
2. www.freejavaguide.com
3. www.java-made-easy.com
4. www.tutorialspoint.com
5. www.roseindia.net

❑ ❑ ❑

Notes

Notes

Chapter : 10

Applet Programming

10.0 OBJECTIVES

Dear Students,

After Studying this chapter you will be able to

- discuss what are applets
- discuss applet architecture and the method of running applets
- discuss a set of methods which provide the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution.
- discribe some basic applet display methods
- explain passing parameters to applets
- explain how to load data into an applet

10.1 INTRODUCTION

Applets are small applications that are accessed on an Internet server, transported over the Internet, automatically installed and run as part of a web document. Once an applet arrives on the client machine it has limited access to resources. Therefore it can produce an arbitrary multimedia user interface and run complex computations without introducing the risk of viruses or breaching data integrity.

10.2 FUNDAMENTALS

To understand applets let us begin with the following example :

```
import java.awt.*;
import java.applet.*;
public class AppletDemo extends Applet {
        public void paint(Graphics g) {
                g.drawstring(“Applet Fundamentals”,
50,20);
        }
}
```

From this example you can see that the applet begins with two **import** statements. The first imports the **Abstract Window Toolkit** (AWT) classes. Applets interact with the user through the AWT and not through console based I/O classes. The AWT contains support for a window based graphical interface. (AWT will be discussed in detail in later chapters). The second **import** statement is **import java.applet.***; This statement imports the applet package which contains the **Applet** class. Every applet that you create must be a subclass of **Applet**.

In the next line we declare a class called AppletDemo which extends the **Applet** class. Remember that this class must be declared as **public**, because it will be accessed by code that is outside the program.

paint() method declared in the class is a method defined by AWT and must be overridden by the applet. Each time the applet wants to redisplay its output, **paint()** should be called. Output may be required to be redisplayed for a number of reasons :

- the window in which the applet is running may be overwritten by another window and then uncovered again
- the applet window may be minimized and then restored

The **paint()** method is also called when the applet begins execution. **paint()** has one parameter **Graphics**. This contains the graphics context which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

drawString() is called inside **paint()**. **drawString()** is a member of the **Graphics** class. The general form of this method is :

void drawString(String *message*, int *x*, int *y*)

This method outputs a string (in this case message) beginning at the specified x, y location. Remember that the upper left hand corner is location 0,0 in Java. Thus the message Applet Fundamentals will be displayed at location 50, 20.

Note that the applet does not have a **main()** method. Instead, an applet begins execution when the name of its class is passed to an appletviewer or to a network browser.

Compile the source code in the same way as you compile other programs. However note the two ways to run an applet :

i) Execute the applet with a Java compatible web browser like Netscape Navigator : To accomplish this you need to write a short HTML text file which contains the appropriate APPLET tag.

To execute AppletDemo the HTML file you write should be as follows :

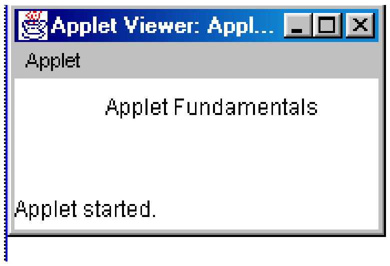
```
<applet code = "AppletDemo" width = 200 height = 60>
</applet>
```

The width and height specify the dimensions of the display area used by the applet. (The APPLET tag contains many other options which we shall discuss subsequently.) After you create this file, you can execute your browser and then load this file. It will cause AppletDemo to be executed.

ii) Use an applet viewer such as the standard JDK tool, appletviewer. An appletviewer executes the applet in a window. This is generally the fastest and easiest way to test your applet.

You can execute the above HTML file in an appletviewer also. eg. if you name your HTML file as ApDemo.html then type the following command line to run the applet :

```
C:\> appletviewer ApDemo.html
```



```
Applet Fundamentals
Applet started.
```

There is another convenient way to do this. In this method, you include a comment at the head of your Java source file that contains the APPLET tag. This documents your code with a prototype of the necessary HTML statements, and you can test your applet by starting the appletviewer with the Java sourcecode file. The revised code of your source program in this situation will be as follows :

```
import java.awt.*;
import java.applet.*;
/* <applet code = "AppletDemo" width = 200 height = 60>
</applet>
*/
public class AppletDemo extends Applet {
        public void paint(Graphics g) {
                g.drawstring(“Applet Fundamentals”,
    50,20);
        }   }
```

Remember :

- *Applets do not need **main()** method*
- *Applets must run under an Appletviewer or a Java compatible browser*
- *User I/O is not accomplished with Java’s stream I/O classes. Instead the interface provided by the AWT is used by applets for user I/O.*

10.1 & 10.2 Check Your Progress.

1. Write True or False.

a) Applets can produce an arbitrary multimedia user interface without introducing the risk of viruses .

b) Applets contain the **main()** method. ....

c) Applets interact with the user through console based I/O classes.

d) Applets can run on a Java compatible browser.

10.3 THE APPLET CLASS

The Applet class provides the necessary support for applet execution. It also provides methods that load and display images, and methods that load and display audio clips. **Applet** extends the **AWT** class **Panel** which in turn extends **Container**, which extends **Component.** These classes provide support for Java’s window based graphical interface.

10.3.1 Applet Architecture

An applet is a window based program. Applets are event driven. An applet resembles a set of interrupt driven routines. An applet waits until an event has occured. The AWT notifies that applet about an event by calling an event handler that has been provided by the applet. Once, this happens the applet must take the necessary action and quickly return control to the AWT. Remember, your applet should not enter a mode of operation in which it maintains control for an extended period. Instead it must perform specific actions in response to events and return control to AWT run time system. If there are situations where your applet needs to perform repetitive tasks on its own, you must start an additional thread of execution.

The user initiates an interaction with the applet, the applet does not. These user interactions are sent to the applet as events to which the applet responds. eg. when the user clicks a mouse within an applet's window, a mouse-clicked event is generated. Applets can contain various controls like push buttons and checkboxes. When the user interacts with these controls an event is generated.

10.3.2 An Applet Skeleton

Applets override a set of methods which provide the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution. Four of these methods **init()**, **start()**,**stop()** and **destroy()** are defined by **Applet**. The method **paint()** is defined by the AWT **Component** class. Default implementations of these methods are provided. Also applets need not override those methods which they do not use.

Let us study these methods with the following skeleton of an applet program :

```
// An Applet Skeleton
import java.awt.*;
import java.applet.*;
/*
<applet code = “Skeleton” width = 300 height = 60>
</applet>
*/

public class Skeleton extends Applet {
    public void init() {
            // initialisation
    }
    public void start() {
            // start or resume execution
    }
    public void stop() {
            // suspends execution
    }
    public void destroy() {
            // perform shut down activity
    }
    public void paint(Graphics g) {
            // redisplay the contents of a window
    }
}
```

When an applet begins, the **AWT** calls the following methods in the given sequence :

init() : This is the first method to be called. This is where variables should be initialised. This method is called only once during the run time of your applet.

start() : The **start()** method is called after **init()**. It is also called to restart an applet after it has been stopped. **start()** is called each time an applet's HTML document is displayed on screen. This means that if an user leaves a web page and comes back, the applet resumes execution at **start()**.

paint() : This method is called everytime the output is to be redrawn. We have already studied **paint()** in the previous section.

When an applet is terminated the following methods are called in this sequence :

stop() : This method is called when a web browser leaves the HTML document containing the applet. When **stop()** is called the applet is probably still running. **stop()** should be used to suspend threads which don't need to run when the applet is not visible. You can restart them when **start()** is called.

destroy() : This method is called when the environment determines that your applet needs to be removed completely from memory. At this point all the resources that the applet may be using should be freed. The **stop()** method is always called before **destroy()**.

update() : update() is a method defined by **AWT**. In some situations you may be required to override the **update()** method. This method is called when your applet has requested that a portion of its window be redrawn. The default version of **update()** first fills an applet with the default background colour and then calls **paint()**. If you fill the background using a different colour in **paint()** the user will experience a flash of the default background everytime **update()** is called. To avoid this, you can override the **update()** method so that it will perform the necessary activities. Then call **update()** in **paint()**. eg.

```
public void update(Graphics g) {
// redisplay your window
}
public void paint(Graphics g) {
update(g);
}
```

10.3 Check Your Progress.

1. Fill in the blanks.

a) .... is the first method to be called in an applet program.

b) .... method is always called before destroy().

c) .... method is called everytime the output is to be redrawn.

2. Write True or False.

a) When **stop()** is called the applet is probably still running.

b) Applets are not event driven.

10.4 SIMPLE APPLET DISPLAY METHOD

In this section let us study some simple applet display methods.

To set the background color of an applet's window use :

setBackground(Color *newColor*)

To set the foreground color use

setForeground(Color *newColor*)

Both these methods are defined by **Component**. *newColor* specifies the new color. The class **Color** defines the constants used to specify colors.

These are :

Color.black	Color.blue	Color.cyan	Color.darkGray
Color.gray	Color.green	Color.lightGray	Color.magenta
Color.orange	Color.pink	Color.red	Color.white
Color.yellow			

You can set colors as : setForeground(Color.cyan);

setBackground(Color.green); etc

The default foreground color is black. The default background color is light gray. Foreground and background colors should generally be set in the **init()** method. They can also be changed as often as necessary during the execution of the applet.

The current settings of the background and foreground colors can be obtained by the following methods respectively :

Color getBackground();

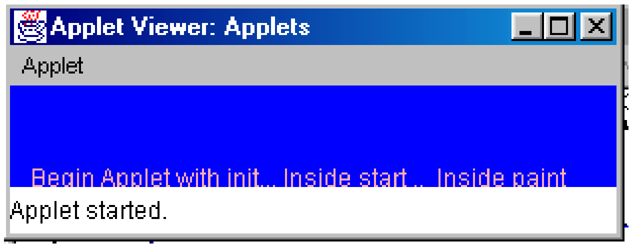
Color getForeground();

Let us study the methods to be used in applets as well as the simple methods described above with the help of the following example :

```
import java.awt.*;
import java.applet.*;
/*
<applet code = "Applets" width = 300, height = 50>
</applet>
*/
public class Applets extends Applet {
    String s1;
    public void init() {
            setBackground(Color.blue);
            setForeground(Color.pink);
```

```
                s1 = “Begin Applet with init...”;
                }
        public void start() {
                s1 += “ Inside start .. “);
                }
        public void paint(Graphics g)     {
                s1+= “ Inside paint “;
                g.drawString(s1, 10,50);
        }
}
```

stop() and **destroy()** are not needed in this applet, we have not overridden them. A sample output of the above program is :



```
Begin Applet with init... Inside start .. Inside paint
Applet started.
```

10.5 REQUESTING REPAINTING

As a general rule, an applet writes to its window only when its **update()** or **paint()** method is called by AWT. Whenever your applet needs to update the information displayed in its window, it simply calls **repaint()**. The **repaint()** method is defined by the **AWT**. It causes the **AWT** run time system to execute a call to your applet's **update()** method. Thus, for another part of your applet, to output to its window, simply store the output and then call **repaint()**. The **AWT** then executes a call to **paint()** which can display the stored information. The **repaint()** method has the following forms :

void repaint()

This version causes the entire window to be repainted.

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

Here, the coordinates of the upper left corner are specified by *left* and *top*, and the *width* and *height* specify the width and height of the region. Remember that these dimensions are specified in pixels. If you only need to update a small portion it is efficient to use **repaint()** only for that region with the second form of paint ().

Note that if your system is slow or busy, **update()** might not be called immediately. Such situation can cause problems especially in cases like animation where a consistent update time is necessary. For this purpose the following forms of **repaint()** can be used :

void repaint(long *maxDelay*)

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

maxDelay specifies the maximum number of milliseconds that can elapse before **update()** is called.

10.4 & 10.5 Check Your Progress.

1. Answer the following.

a) Write the methods used to set the Background and Foreground colors in an applet.

____

____

b) Which is the default foreground color?

____

____

c) What is the use of the **repaint()** method.

____

____

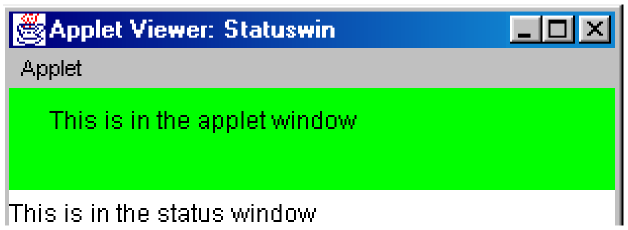
10.6 USING THE STATUS WINDOW

In addition to displaying information in its window, an applet can also output a message to the status window of the browser or applet viewer on which it is running. For this purpose, you make use of **showStatus()** with the string that you want to display. The status window can be used to give the user a feedback about what is occuring in the applet, suggest options or possibly report errors.

The following example demonstrates the use of **showstatus()** :

```
import java.awt.*;
import java.applet.*;
/*
<applet code = “Statuswin” width = 300 height = 50>
</applet>
*/
public class Statuswin extends Applet {
    public void init() {
            setBackground(Color.green);
    }
    public void paint(Graphics g) {
```

```
                    g.drawString("This is in the applet window ",
20, 20);
                    showStatus("This is in the status window");
            }
    }
    output
```



```
This is in the applet window
This is in the status window
```

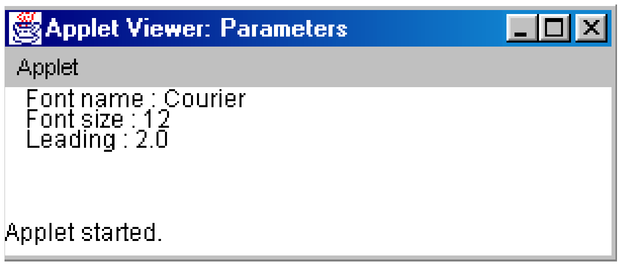
10.7 PASSING PARAMETERS TO APPLETS

The **APPLET** tag in HTML allows you to pass parameters to your applet. To retrieve a parameter, you have the **getParameter()** method, which returns the value of the specified parameter in the form of a **String** object. Therefore numeric and **Boolean** values have to be converted from their string representations to their internal formats. The following example will illustrate :

```
import java.awt.*;
import java.applet.*;
/*
<applet code = "Parameters" width = 300 height = 60>
<param name = fontName value = Courier>
<param name = fontSize value = 12>
<param name = leading value = 2>
</applet>
*/
public class Parameters extends Applet {
String fontName;
int fontSize;
float leading;
public void start() {
        String p;
        fontName = getParameter("fontName");
        if(fontName == null)
                fontName = "Not found";
        p = getParameter("fontSize");
        try {
                if(p != null)
                        fontSize = Integer.parseInt(p);
                else
                        fontSize = 0;
                }
                catch(NumberFormatException e) {
                        fontSize = -1;
                }
                p = getParameter("leading");
                try {
                if(p != null)
                        leading = Float.valueOf(p).floatValue();
                else
                        leading = 0;
                }
                catch(NumberFormatException e) {
                        leading = -1;
                }
        }
        public void paint(Graphic g) {
                g.drawString("Font name : " + fontName, 10, 10);
                g.drawString("Font size : " + fontSize, 10, 20);
                g.drawString("Leading : "+ leading 10, 30);
        }
    }
```

output

It is important to note that in this program we have checked the return values from **getParameter()**. If a parameter is not available, **getParameter()** returns a **null**. Also note that the conversion from **String** type to numeric is done in a **try** statement which catches the **NumberFormatException**. There should be no uncaught exceptions in an applet.



```
Font name : Courier
Font size : 12
Leading : 2.0
```

Applet started.

10.8 LOADING DATA INTO APPLET

getDocumentBase() and **getCodeBase()** functions are used to explicitly load media and text from applets. Java allows the applet to load data from the directory holding the HTML file which started the applet (the document base) and the directory from which the applet's class file was loaded (the codebase). These directories are returned as URL objects by **getDocumentBase()** and **getCodeBase()** respectively. They can be concatenated with a string which names the file you want to load. To actually load another file, you use the **showDocument()** method which is defined by the **AppletContext** interface. **AppletContext** is an interface which lets you get information from the applet's execution environment. Once you have obtained the applet's context, you can bring another document into view by calling **showDocument()**. This method has no return values and it does not throw any exception if it fails.

There are two **showDocument()** methods :

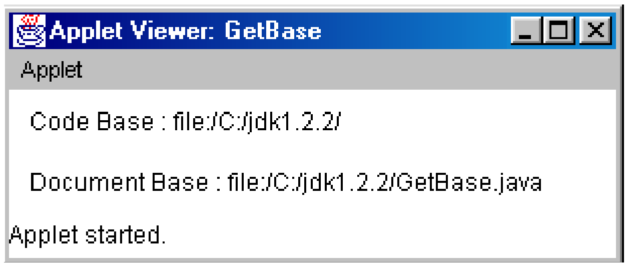
showDocument(URL) : displays the document specified by URL.

showDocument(URL, *where*) : displays the document at the location specified within the browser window by *where*. The valid arguments for where are : _self (current frame), _parent (parent frame), _top (topmost frame), _blank (new browser window).

The following program shows how to use the **getDocumentBase()** and **getCodeBase()** :

```
import java.awt.*;
import java.applet.*;
import java.net.*;
/*
<applet code = "GetBase" width = 300 height = 60 >
</applet>
*/
public class GetBase extends Applet {
        public void paint(Graphics g) {
                String s1;
                URL url = getCodeBase();
                s1 = "Code Base : " + url.toString();
                g.drawString(s1, 10, 20);
                url = getDocumentBase();
                s1 = "Document Base : " + url.toString();
                g.drawString(s1, 10, 40);
        }
}
```

The output of the program will be :



```
Code Base : file:/C:/jdk1.2.2/

Document Base : file:/C:/jdk1.2.2/GetBase.java

Applet started.
```

The **AudioClipInterface** defines the methods : **play()** - play a clip from the beginning, **stop()** - stop playing the clip, and **loop()** - play the loop continuously. These methods can be used to play audio clips once you have loaded an audio clip with the **getAudioClip()** method.

The **AppletStub** interface provides the means by which an applet and the browser or appletviewer communicate. This interface is normally not implemented in your code.

10.6 to 10.8 Check Your Progress.

1. Fill in the blanks.

a) The .... window can be used to give the user a feedback about what is occuring in the applet.

b) The .... tag in HTML allows you to pass parameters to your applet.

c) .... and .... functions are used to explicitly load media and text from applets.

10.9 SUMMARY

Applets are small applications that are accessed on an Internet server, transported over the Internet, automatically installed and run as part of a web document. Once an applet arrives on the client machine it has limited access to resources. Therefore it can produce an arbitrary multimedia user interface and run complex computations without introducing the risk of viruses or breaching data integrity.

Applets override a set of methods which provides the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution. Some of these methods init(), start(),stop(), paint() and destroy() are defined by Applet.

Source : *www.scribd.com(Link)*

10.10 CHECK YOUR PROGRESS - ANSWERS

10.1 & 10.2

1. a) True b) False c) False

d) True

10.3

1. a) init()

b) stop()

c) paint()

2. a) True

b) False

10.4 & 10.5

1. a) setBackground(Color *newColor*) and setForeground(Color *newColor*) are the methods to set the Background and Foreground colors in an applet.

b) The default foreground color is black.

c) Whenever an applet needs to update the information displayed in its window it makes use of the **repaint()** method.

10.6 to 10.8

1. a) status

b) APPLET

c) getDocumentBase(), getCodeBase()

10.11 QUESTIONS FOR SELF - STUDY

1. What is the **paint**() method? Give reasons why an output may be required to be repainted?
2. Describe the ways to run an applet.
3. Describe the skeleton of an applet program.
4. Describe some simple applet display methods.
5. **Write short notes on :**

   a) Applet Architecture

   b) The repaint() method

   c) Loading Data into an Applet
6. What is the use of Status window? Describe with example the **showStatus**() method.

10.12 SUGGESTED READINGS

1. www.java.com
2. www.freejavaguide.com
3. www.java-made-easy.com
4. www.tutorialspoint.com
5. www.roseindia.net

❑ ❑ ❑

Notes

Notes

Chapter : 11

Event Handling

11.0 OBJECTIVES

Dear Students,

After studying this chapter you will able to :

- Explain the meaning of what is an event
- Explain the Delegation Event Model
- discuss some important Event Classes
- discuss some sources of events and the event listener interfaces
- discuss some sample programs for handling simple events.

11.1 INTRODUCTION

An important aspect of Java that relates to applets is **events**. Applets are event driven programs. The most commonly handled events are those generated by the mouse, the keyboard, and the various controls like push buttons. Events are supported by the **jawa.awt.event** package.

11.2 THE DELEGATION EVENT MODEL

The delegation event model defines standard and consistent mechanisms to generate and process events. In this model, the concept is, that a source generates an event and sends it to one or more listeners. The listener simply waits for an event, once it receives the event, the listener processes it and returns. The advantage of this design is that the application logic which processes the events is cleanly separated from the user interface logic, that generates the events. Thus the user interface element is able to delegate the processing of an event to a separate piece of code. In this model, the listeners must register with a source in order to receive the notification of an event. This means that notifications are sent only to those listeners who wish to receive them.

11.2.1 Events

In the delegation model, an event is an object which describes a state change in a source. The event can be generated as a result of a person interacting with the elements in a graphical user interface. Pressing a button, entering a character via a keyboard, selecting an item in a list, clicking the mouse are some of the activities which cause events to be generated. Events may also occur as a consequence which is not directly caused by user interface. eg. an event may be generated if a timer expires, a counter exceeds a value, a hardware or software failure occurs etc. You are free to define events that are appropriate to your application.

11.2.2 Sources of Events

A source is an object which generates an event. This occurs when the internal state of the object changes in some way. Sources may generate more than one type of event. A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method.

The general form is :

public void add*Type*Listener(*Type*Listener *el*)

where *Type* is the name of the event and *el* is the reference to the event listener. eg the method that registers a keyboard event listener is called **addKeyListener()**.

When an event occurs, all the registered listeners are notified and receive a copy of the event object. This is known as ***multicasting*** the event.

Some sources may allow only one listener to register.

The general form of this method is :

public void addTypeListener(*Type*Listener *el*)

throws java.util.TooManyListenersException

where *Type* is the name of the event and *el* is the reference to the event listener. When such an event occurs, the registered listener is notified. This is called as **unicasting** the event.

A source must also provide a method to allow a listener to unregister an interest in a specific event. The general form of this method is :

public void remove *Type*Listener(*Type*Listener *el*)

here, *Type* is the name of the event and *el* is the reference to the event listener eg to remove a keyboard listener you would call the **removeKeyListener()** method.

The methods that add or remove listeners are provided by the source that generates the events.

11.2.3 Event Listeners

A listener is an object that is notified when an event occurs. It has two major requirements :

- First it must have been registered with one or more sources to receive the notifications about specific types of event.
- Second it must implement methods to receive and process these notifications.

The methods which receive and process events are defined in a set of interfaces found in **java.awt.event**.

11.1 & 11.2 Check Your Progress.

1. Fill in the blanks.

a) Events are supported by the .... package.

b) In the delegation model, an .... is an object which describes a state change in a source.

c) A .... is an object which generates an event.

d) A .... is an object that is notified when an event occurs.

2. Write True or False.

a) Pressing a button causes events to be generated.

b) A listener need not have been registered with one or more sources to receive the notifications about specific types of event.

11.3 EVENT CLASSES

11.3.1 Introduction

The classes which represent events are at the core of Java's event handling mechanism. At the root of the Java class event hierarchy is the **EventObject**, which is in **java.util**. It is the superclass for all events. One of its constructors is :

EventObject(Object *src*)

where *src* is the object that generates the event.

EventObject contains two methods : **getSource()** and **toString()**. **getSource()** method returns the source of the event. Its general form is :

Object getSource()

toString() returns the string equivalent of the event.

The class **AWTEvent**, which is defined in the **java.awt package**, is a subclass of **EventObject**. It is the superclass of all AWT-based events used by the delegation event model. The method **getID()** of this class is used to determine the type of the event.

Its form is :

int getID()

The package **java.awt.event** defines several types of events that are generated by various user interface elements. Some of the most important of these event classes are given in the table below :

Event Class	**Description**
Action Event	Generated when a button is pressed, a list item is double clicked or a menu item is selected
Adjustment Event	Generated when a scrollbar is manipulated Component Event Generated when a component is hidden, moved, resized or becomes visible
Container Event	Generated when a component is added to or removed from a container Focus Event Generated when a component gains or loses keyboard focus

Input Event	Abstract superclass for all component input event classes
	Item Event Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected
Key Event	Generated when input is received from the keyboard
Mouse Event	Generated when the mouse is dragged, moved, clicked, pressed, or released, also generated when the mouse enters or exits a component
Text Event	Generated when the value of a text area or text field is changed
Window Event	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened or quit

11.3.1 Check Your Progress.

1. Answer the following.

a) List the methods of the **EventObject** class.

____

____

b) Describe the following event classes :

(i) ActionEvent : ____

____

ii) Mouse Event : ____

____

Let us briefly go through some of the classes listed in the previous section and some methods of these classes :

11.3.2 The Action Event Class

From the table we can see that an **ActionEvent** is generated when a button is pressed, a list item is double clicked or a menu item is selected. This class defines four integer constants which can be used to identify any modifiers associated with an action event. These constants are **ALT_MASK**, **CTRL_MASK**, **META_MASK**, **SHIFT_MASK**. There is also an integer constant **ACTION_PERFORMED**, which can be used to identify action events.

The constructors of **ActionEvent** are :

ActionEvent(Object *src*, int *type*, String *cmd*)

ActionEvent(Object *src*, int *type*, String *cmd*, int *modifiers*)

where *src* is a reference to the object which generated the event. *type* specifies the type of the event and *cmd* is the command string. The argument *modifiers* indicates which of the modifier keys (ALT, CTRL, META and/or SHIFT) were pressed when the event was generated.

getActionCommand() is the method which used to obtain the command name for the invoking **ActionEvent** object. Its form is :

String getActionCommand()

The **getModifiers()** method returns a value that indicates which modifier keys were pressed when the event was generated. Its general form is :
int getModifiers()

11.3.3 The Component Event Class

This class is generated when the size, position or visibility of a component is changed. There are four types of component events. **The ComponentEvent** class defines integer constants that can be used to identify these component events. The constants and their meanings are :

COMPONENT_HIDDEN	The component was hidden
COMPONENT_MOVED	The component was moved
COMPONENT_RESIZED	The component was resized
COMPONENT_SHOWN	The component became visible

The constructor for the class is :

ComponentEvent(Component *src*, int *type*)

where *src* is a reference to the object that generated this event and *type* specifies the type of the event.

ComponentEvent is the superclass directly or indirectly of **ContainerEvent**, **FocusEvent**, **KeyEvent**, **MouseEvent** and **WindowEvent**. Its method **getComponent()** returns the component which generated the event and its general form is :

Component getComponent()

11.3.4 The Container Event Class :

A **ContainerEvent** is generated when a component is added to or removed from a container. There are two types of container events. The class defines integer constants that can be used to identify them : **COMPONENT_ADDED** and **COMPONENT_REMOVED**. They indicate that a component has been added or removed from the container.

ContainerEvent is a subclass of **ComponentEvent**. Its constructor is :

ContainerEvent(Component *src*, int *type*, Component *comp*)

where *src* is a reference to the container that generated this event, the type of the event is specified by *type*, and the component that has been added or removed is specified by *comp*.

With the **getContainer()** method you can obtain a reference to the container that generated this event. The method has the general form :

Container getContainer()

The **getChild()** is the method which returns a reference to the component that was added to or removed from the container. Its general form is :

Component getChild()

11.3.2 to 11.3.4 Check Your Progress

1. Fill in the blanks.

a) .... is the method which used to obtain the command name for the invoking **ActionEvent** object.

b) The .... class is generated when the size, position or visibility of a component is changed.

c) The two constants used to identify container events are .... and ....

11.3.5 The Input Event Class

This abstract class is a subclass of **ComponentEvent** and is the superclass for component input events. **KeyEvent** and **MouseEvent** are its subclasses. The **InputEvent** class defines the following eight integer constants that can be used to obtain information about any modifiers associated with this event :

ALT_MASK	**BUTTON2_MASK**	**META_MASK**
ALT_GRAPH_MASK	**BUTTON3_MASK**	**SHIFT_MASK**
BUTTON1_MASK	**CTRL_MASK**	

The methods **isAltDown()**, **isAltGraphDown()**, **isControlDown()**, **isMetaDown()** and **isShiftDown()** are used to test whether these modifiers were pressed at the time this event was generated. All these methods return a boolean value.

The **getModifiers()** method returns a value that contains all of the modifier flags for this event. Its form is :

int getModifiers()

11.3.6 The Item Event Class

An ItemEvent is generated when a check box or list item is clicked or when a checkable menu item is selected or deselected. There are two types of item events and they are identified by the following integer constants :

DESELECTED	The user deselects an item
SELECTED	The user selects an item

This class also defines one integer constant **ITEM_STATE_CHANGED** which signifies a change of state. **ItemEvent** has the following constructor :

ItemEvent(ItemSelectable *src*, int *type*, Object *entry*, int *state*)

src is a reference to the component that generated the event, *type* specifies the type of the event. The specific item that generated the item event is passed in *entry*, and the current state of that item is passed in *state*.

In order to obtain a reference to the item that generated an event we use the method **getItem()**. Its general form is :

Object getItem()

The **getStateChanged()** method returns the state changed for the event.

Its form is :

int getStateChanged()

where it returns **SELECTED** or **DESELECTED**.

The **getItemSelectable()** method is used to obtain a reference to the **ItemSelectable** object that generated the event. Its form is :

ItemSelectable getItemSelectable()

11.3.7 The Key Event Class

A **KeyEvent** is generated when a keyboard input occurs. There are three types of key events which are identified by the integer constants : **KEY_PRESSED**, **KEY_RELEASED**, **KEY_TYPED**. The first two events are generated when any key is pressed or released. The last event occurs only when a character is generated. We know that all key presses do not result in characters eg. pressing the SHIFT key will not generate any character.

KeyEvent defines many other integer constants. eg. **VK_0** through **VK_9** and **VK_A** through **VK_Z** define the ASCII equivalents of the numbers and literals respectively. Some other constant are :

where the **VK** constants specify the virtual key codes and are independant of any modifiers like control, shift or alt.

KeyEvent is a subclass of **InputEvent**. Its constructors are :

KeyEvent(Component *src*, int *type*, long *when*, int *modifiers*, int *code*)

KeyEvent(Component *src*, int *type*, long *when*, int *modifiers*, int *code*, char *ch*)

where *src* is a reference to the component that generated the event, type of the event is specified by *type*. The system time at which the key was pressed is passed in *when*. The *modifiers* argument indicates which modifiers were pressed when this key event occured. The virtual key code is passed in *code*. The character equivalent, if it exists is passed in *ch*. If no valid character exists, then *ch* contains **CHAR_UNDEFINED**.

The commonly used methods of the **KeyEvent** class are :

getKeyChar() : which returns the character that was entered and has the following general form :

char getKeyChar()

getKeyCode() : which returns the key code. Its general form is :

int getKeyCode()

11.3.5 to 11.3.7 Check Your Progress.

1. Fill in the blanks.

a) .... and .... are the subclasses of the InputEvent class.

b) The two types of item events are identified by the integer constants .... and ....

c) A.................. is generated when a keyboard input occurs.

d) There are three types of key events which are identified by the integer constants : ...., ... and ....

11.3.8 The Mouse Event Class :

There are seven types of mouse events. The following integer constants are used to identify them :

MOUSE_CLICKED	**MOUSE_DRAGGED**	**MOUSE_ENTERED**
MOUSE_EXITED	**MOUSE_MOVED**	**MOUSE_PRESSED**
MOUSE_RELEASED		

MouseEvent is a subclass of **InputEvent** and has the following constructor:

MouseEvent(Component *src*, int *type*, long *when*, int *modifiers*,
int *x*, int *y*, int *clicks*, boolean *triggersPopup*)

Here *src* is a reference to the component that generated the event. The type of the event is specified by *type*. *when* specifies the system time at which the mouse event occured. The *modifiers* argument indicates which modifiers were pressed when a mouse event occured. The coordinates of the mouse are passed in *x* and *y*. *clicks* specifies the click count. The *triggersPopup* flag indicates if this event causes a pop up menu to appear on this platform.

The most commonly used method of this class are **getX()** and **getY()** which return the coordinates of the mouse when the event occured. Their general forms are :

int getX()

int getY()

The **getPoint()** method returns the coordinates of the mouse. This method has the following form:

Point getPoint()

It returns a **Point** object, which contains the X, Y coordinates in its integer members x and y.

translatePoint() method changes the location of the event. Its form is :

void translatePoint(int *x*, int *y*)

The arguments *x* and *y* are added to the coordinates of the event.

11.3.9 The Window Event Class

There are seven types of window events and the **WindowEvent** class defines integer constants to identify them. The constants are :

WINDOW_ACTIVATED	**WINDOW_CLOSED**
WINDOW_CLOSING	**WINDOW_DEACTIVATED**
WINDOW_DEICONIFIED	**WINDOW_ICONIFIED**
WINDOW_OPENED	

WindowEvent is a subclass of **ComponentEvent** and its constructor is :

WindowEvent(Window *src*, int *type*)

where *src* is a reference to the object that generated this event and *type* is the type of the event.

getWindow() is the most commonly used method of this class and it returns the **Window** object that generated this event. Its general form is :

Window getWindow()

11.3.8 & 11.3.9 Check Your Progress.

1. Fill in the blanks.

a) There are .... types of mouse events.

b) .... is the most commonly used method of the WindowEvent class.

11.4 SOURCES OF EVENTS

The table given on the next page lists some of the user interface components that can generate the events described in the previous section. In addition to these, other components like applets can also generate events. You can also build your own components that generate events.

Event Source	Description
Button is	Generates action events when the button pressed
Checkbox box	Generates item events when the check is selected or deselected
Choice	Generates item events when the choice is changed
List	Generates action events when an item is double clicked, generates an item event when an item is selected or deselected
Menu Item	Generates action events when a menu item is selected, generates item events when a checkable menu item is selected or deselected
Scrollbar	Generates adjustment events when the scroll bar is manipulated
Text components	Generates text events when the user enters a character
Window	Generates window events when a window is activated, closed, deactivated, diconified, iconified, opened or quit

11.5 EVENT LISTENER INTERFACES

Listeners are created by implementing one or more of the interfaces defined by the **java.awt.event** package. When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its argument. Let us study some of the listener interfaces and the specific methods contained in them in this section :

The ActionListener Interface :

This interface defines one method to receive action events. The method is **actionPerformed()** and is invoked when an action event occurs. Its general form is :

void actionPerformed(ActionEvent *ae*)

The ComponentListener Interface :

This interface defines four methods that are invoked when a component is resized, moved, shown or hidden. The general forms of these methods are :

void ComponentResized(ComponentEvent *ce*)

void ComponentMoved(ComponentEvent *ce*)

void ComponentShown(ComponentEvent *ce*) void ComponentHidden(ComponentEvent *ce*)

The ContainerListener Interface :

This interface contains two methods **componentAdded()** and **componentRemoved()**. **componentAdded()** is invoked when a component is added to the container, and **componentRemoved()** is invoked when the component is removed from the container. The general forms of these methods are :

void ComponentAdded(ContainerEvent *ce*)

void ComponentRemoved(ContainerEvent *ce*)

ItemListener Interface :

This defines the i**temStateChanged()** method that is invoked when the state of an item changes. Its general form is :

void itemStateChanged(ItemEvent *ie*)

The KeyListenerInterface : This interface defines three methods

- **keyPressed()** and **keyReleased()** are invoked when a key is pressed and released respectively. The general forms of these methods are :

void KeyPressed(KeyEvent *ke*)   void KeyReleased(KeyEvent *ke*)

- **keyTyped()** method is invoked when a character has been entered.

Its general form is :

void keyTyped(KeyEvent *ke*)

The MouseListenerInterface :

This interface defines five methods. If the mouse is pressed and released at the same point, **mouseClicked()** is invoked. Its general form is :

void mouseClicked(MouseEvent *me*)

When the mouse enters a component the **mouseEntered()** method is called. Its form is :

void mouseEntered(MouseEvent *me*)

When it leaves a component the **mouseExited()** is called. Its general form is :

void mouseExited(MouseEvent *me*)

The **mousePressed()** and **mouseReleased()** methods are invoked when the mouse is pressed and released respectively. Their general forms are :

void mousePressed(MouseEvent *me*)

void mouseReleased(MouseEvent *me*)

The MouseMotionListener Interface :

This interface defines two methods. The **mouseDragged()** is called multiple times as the mouse is dragged. The **mouseMoved()** method is called multiple times as the mouse is moved. Their general forms are :

void mouseDragged(MouseEvent *me*)

void mouseMoved(MouseEvent *me*)

The WindowListener Interface :

The interface defines seven methods. The **windowActivated()** and **windowDeactivated()** methods are invoked when a window is activated or deactivated respectively. If a window is iconified, the **windowIconified()** method is called and when it is deiconified, the **windowDeiconified()** method is called. When a window is opened or closed, the **windowOpened()** or **windowClosed()** methods are called, respectively. The **windowClosing()** method is called when a window is being closed. The general forms of these methods are:

void windowActivated(WindowEvent *we*)

void windowClosed(WindowEvent *we*)

void windowClosing(WindowEvent *we*)

void windowDeactivated(WindowEvent *we*)

void windowDeiconified(WindowEvent *we*)

void windowIconified(WindowEvent *we*)

void windowOpened(WindowEvent *we*)

11.5 Check Your Progress.

1. Match the following.

Column A	**Column B**
a) actionPerformed()	(i) MouseMotionListener interface
b) componentAdded()	(ii) ItemListener interface
c) mouseDragged()	(iii) ContainerListener interface
d) mouseClicked()	(iv) ActionListener interface
e) itemStateChanged()	(v) MouseListener interface

11.6 HANDLING EVENTS

In this section, we shall study examples which will handle the two most commonly used event generators : the keyboard and the mouse. We shall use the delegation event model in applet programming examples. The steps are :

- Implement the appropriate interface in the listener so that it will receive the type of the event desired.

- Implement the code to register and unregister (if required) the listener as a recipient for the event notifications.

Remember that a source may generate several types of events. Each event must be registered separately. Also, an object may register to receive several types of events and therefore it must implement all of the interfaces that are required to receive these events.

11.6.1 Handling Keyboard events

The following example will demonstrate how to handle simple keyboard events. Let us briefly review how the keyboard events are generated. When a key is pressed a **KEY_PRESSED** event is generated and the **keyPressed()** event handler is called. Similarly when a key is released, a **KEY_RELEASED** event is generated and the **keyReleased()** handler is executed. If a keystroke generates a character then the **KEY_TYPED** event is generated and the **keyTyped()** handler invoked. Thus, with every key press at least two or many times three events are generated. If your program needs to handle special keys like arrow or function keys then the **keyPressed()** handler should be used. An important point to remember is that in order for your program to receive keyboard events, it must request input focus. For this purpose, call the **requestFocus()** which is defined by **Component**. Else, no keyboard events will be received.

```
        // Demonstrate keyboard events
        import java.awt.*;
        import java.awt.event.*;
        import java.applet.*;
        /*
        <applet code = “KeyEvents” width = 300 height = 60>
        </applet>
        */
        public class KeyEvents extends Applet implements
KeyListener {
            String msg = “ “;
            int X = 10, Y = 40;    // set coordinates to output
            public void init() {
                addKeyListener(this);
                requestFocus();    // request input focus
            }
                public void keyPressed(KeyEvent ke) {
                showStatus(“Key Pressed”);
            }

            public void keyReleased(KeyEvent ke) {
                showStatus(“Key Released”);
            }

            public void keyTyped(KeyEvent ke) {
                msg+= ke.getKeyChar();
                repaint();
```

```
                        }

                        public void paint(Graphics g) {
                              g.drawString(msg, X, Y);                  //
display the key typed
                        }
    }
```

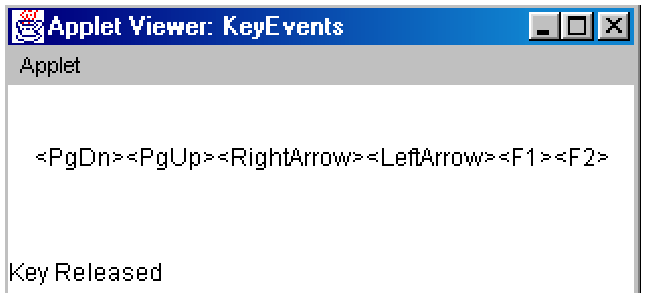
Run this program and check the status when the keyboard keys are pressed and released.

In order to handle special keys like the function keys or the arrow keys you are required to use the **keyPressed()** handler. The **keyTyped()** handler does not provide them. Also to identify these you have to use their virtual key codes. Modify the **keyPressed()** handler of your above program as shown below to illustrate how to respond to special keys.

```
    public void keyPressed(KeyEvent ke) {
                        showStatus("KeyPressed");
                        int k = ke.getKeyCode();
                        switch(k) {
                              case KeyEvent.VK_PAGE_UP:
                                      msg += "<PgUp>";
                                      break;
                              case KeyEvent.VK_PAGE_DOWN:
                                      msg += "<PgDn>";
                                      break;
                              case KeyEvent.VK_LEFT:
                                      msg += "<LeftArrow>";
                                      break;
                              case KeyEvent.VK_RIGHT:
                                      msg += "<RightArrow>";
                                      break;
                              case KeyEvent.VK_F1 :
                                      msg += "<F1>";
                                      break;
                              case KeyEvent.VK_F2 :
                                      msg += "<F2>";
                                      break;
                              }
                        repaint();
```

```
}
```

Sample output of the above program is as shown below :



```
<PgDn><PgUp><RightArrow><LeftArrow><F1><F2>

Key Released
```

11.6.2 Handling Mouse Events

For handling mouse events you have to implement the **MouseListener** and the **MouseMotionListener** interfaces. The following program shows the current coordinates of the mouse in the status window of the applet. Everytime a button is pressed the word "Down" is displayed at the current mouse pointer location and everytime a button is released, the word "Up' is shown. If a button is clicked the message "Mouse Clicked" is displayed in the upper left corner of the applet display area. Also whenever a mouse enters or exits the applet window a message is displayed in the upper left hand corner. When a mouse is dragged a * is displayed which tracks the mouse pointer as it is dragged.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code = "MouseEvents" width = 300 height = 60>
</applet>
*/
public class MouseEvents extends Applets implements
MouseListener,
MouseMotionListener {
    String msg = " ";
    int X = 0, Y = 0;          // Mouse coordinates
    public void init() {
        addMouseListener(this);
        addMouseMotionListener(this);
    }
    public void mouseClicked(MouseEvent me) {
        X = 0;
        Y = 10;
        msg = “Mouse Clicked”;
        repaint();
}
public void mouseEntered(MouseEvent me) {
        X = 0;
        Y = 10;
        msg = “Mouse entered”;
        repaint();
}
public void mouseExited(MouseEvent me) {
        X = 0;
        Y = 10;
        msg = “Mouse Exited”;
        repaint();
}
public void mousePressed(MouseEvent me) {
        X = me.getX();
        Y = me.getY();          // get coordinates
        msg = “Down”;
        repaint();
}
public void mouseReleased(MouseEvent me) {
        X = me.getX();
        Y = me.getY();          // get coordinates
        msg = “Up”;
        repaint();
}
public void mouseDragged(MouseEvent me) {
        X = me.getX();
        Y = me.getY();          // get coordinates
        msg = “*”;
        showStatus(“Dragging Mouse” + X+ “,”+ Y);
        repaint();
}
public void mouseMoved(MouseEvent me) {
```

```
                showStatus("Moving Mouse");
            }
            public void paint(Graphics g) {
                g.drawString(msg, X, Y);
            }
        }
```

run above program and study output

Note that the **MouseEvents** class extends **Applet** and implements both **MouseListener** and **MouseMotionListener** interfaces. These interfaces contain methods which receive and process the mouse events. Each method handles its event and then returns.The applet is both the source and the listener of the events, because **Component** which supplies the **addMouseListener()** and **addMouseMotionListener()** methods is a superclass of **Applet**.

11.7 ADAPTER CLASSES

Adapter classes is a special feature of Java. It helps to simplify event handlers in certain situations. An adapter class is a class which provides an empty implementation of all the methods in an event listener interface. Thus these classes are useful in situations when you want to receive and process only some of the events that are handled by a particular event listener interface. You can define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which you are interested. For example, the **Mouse Motion Adapter** class has two methods **mouse Dragged()** and **mouse Moved()** whose signatures are exactly as defined in the **MouseMotionListener** interface. Suppose you are interested only in mouse dragged events, then you can simply extend **Mouse Motion Adapter** and implement **mouse Dragged ()**. The empty implementation of **mouse Moved ()** would handle the mouse events.

The following table shows the different adapter classes in java.awt.event and the interface that each implements.

Adapter Class	Listener Interface
Component Adapter	Component Listener
Container Adapter	Container Listener
Focus Adapter	Focus Listener
Key Adapter	Key Listener
Mouse Adapter	Mouse Listener
Mouse Motion Adapter	Mouse Motion Listener
Window Adapter	Window Listener

11.6 & 11.7 Check Your Progress.

1. Fill in the blanks.

a) In order for your program to receive keyboard events, it must request input focus by calling the .... method.

b) In order to handle special keys like the function keys or the arrow keys you are required to use the .... handler.

c) An .... class provides empty implementations of all methods in an event listener interface.

d) The .... adapter class implements the Window Listener interface.

11.8 SUMMARY

Applets are event driven programs. The most commonly handled events are those generated by the mouse, the keyboard, the various controls like push buttons. Events are supported by the jawa.awt.event package.

The delegation event model defines standard and consistent mechanisms to generate and process events. In the delegation model, an event is an object which describes a state change in a source.

11.9 CHECK YOUR PROGRESS - ANSWERS

11.1 & 11.2

1. a) java.awt.event b) event

c) source d) listener

2. a) True b) False

11.3.1

1. a) **EventObject** contains two methods : **getSource()** and **toString()**.

b) i) ActionEvent : Generated when a button is pressed, a list item is double clicked or a menu item is selected

ii) MouseEvent :Generated when the mouse is dragged, moved, clicked, pressed, or released, also generated when the mouse enters or exits a component

11.3.2 to 11.3.4

1. a) getActionCommand()

b) ComponentEvent

c) COMPONENT_ADDED, COMPONENT_REMOVED

11.3.5 to 11.3.7

1. a) KeyEvent, MouseEvent

b) DESELECTED, SELECTED

c) KeyEvent

d) KEY_PRESSED, KEY_RELEASED, KEY_TYPED

11.3.8 & 11.3.9

1. a) seven

b) getWindow()

11.4

1. a) Generates action events when the button is pressed

b) Generates adjustment events when the scroll bar is manipulated

11.5

1. a) - (iv)

b) - (iii)

c) - (i)

d) - (v)

e) - (ii)

11.6 & 11.7

1. a) requestFocus() c) Adapter

b) keyPressed() d) Window Adapter

11.10 QUESTIONS FOR SELF - STUDY

1. Describe the Delegation Event Model in detail.
2. What do you understand by multicasting and unicasting?
3. Describe in detail any five Event classes and some of the methods of these classes.
4. Describe the various user interface components that can generate events.
5. List the steps to handle event generators using the delegation event model.
6. Describe any four EventListener interfaces and the specific methods contained in them.

11.11 SUGGESTED READINGS

1. www.java.com
2. www.freejavaguide.com
3. www.java-made-easy.com
4. www.tutorialspoint.com
5. www.roseindia.net

❑ ❑ ❑

Notes

Notes

Chapter : 12

Introducing Abstract Window Toolkit

12.0 OBJECTIVES

Dear Students,

After studying this chapter you will able to :

- discuss what is the Abstract Window Toolkit
- discuss how to create frames in applets
- discuss how to draw lines, circles etc.
- to obtain basic knowledge of colors and the **Font** class

12.1 INTRODUCTION

The **AWT** (Abstract Window Toolkit) contains numerous classes and methods which allow you to create and manage windows. In this chapter you will learn how to create and manage windows, manage fonts, output text and utilise graphics. The next chapter will describe the various controls like push buttons and scroll bars supported by AWT. Remember that though the main purpose of the AWT is to support applet windows, it can also be used to create stand alone windows that run in a GUI environment. The AWT classes are contained in the **java.awt** package. It is one of the largest packages of Java. We shall study some of these classes in this and the next chapter. Remember however, that a brief introduction of the various classes and methods only has been presented. A detailed study is not in the scope of this study material.

12.2 FUNDAMENTALS

The AWT defines windows according to a class hierarchy that adds functionality and specificity with each level. The two most common windows are those derived from **Panel**, which is used by applets and those derived from **Frame** which creates a standard window. At the top of the AWT hierarchy is the **Component** class. This is an abstract class which encapsulates all of the attributes of a visual component. All those user interfaces which are displayed on the screen and those which interact with the user are subclasses of **Component**. **Component** defines over a hundred public methods which are useful for managing events like mouse and keyboard inputs, etc. The **Container** class is a subclass of **Component**. It has additional methods which allow other **Component** objects to be nested within it. Other container objects can be stored inside a **Container**, since they themselves are instances of **Component**. A container is responsible for laying out any components that it contains. This is done through the use of **layout managers**.

The **Panel** class is a concrete subclass of **Container**. It simply implements the **Container**. It does not add any new methods. **Panel** is the superclass for **Applet**. When screen output is directed to an applet, it is drawn on the surface of a panel object. A **Panel** is a window which does not contain a title bar, a menu bar or a border. This is the reason you don't see these items when an applet is run inside a browser. However, when you run an applet in an applet viewer, the applet viewer provides the title and the border. Other components can be added to the **Panel** object by using the **add()** method which is inherited from **Container**.

The **Window** class creates a top level window. A top level window is a window which is not contained within any other object; it sits directly on the desktop. Generally, you don't create a **Window** object directly. You use a subclass of **Window** called **Frame**.

Frame is a subclass of **Window** and has a title bar, menu bar,

12.1 & 12.2 Check Your Progress.

1. Fill in the blanks.

a) The AWT classes are contained in the .... package.

b) The two most common windows are those derived from .... and those derived from ....

c) A .... is responsible for laying out any components that it contains.

2. Write True or False.

a) A **Panel** is a window which contains a title bar, a menu bar or a border. ....

b) **Canvas** is a part of the hierarchy for applet or frame windows. ....

borders and resizing corners. If you create a **Frame** object from within an applet, it will contain a warning message (eg. Warning : Applet Window) to the user indicating that an applet window has been created. This message warns the user that the window was started by an applet and not by software running on their computer.

Canvas encapsulates a blank window upon which you can draw. Remember **Canvas** is not a part of the hierarchy for applet or frame windows.

12.3 FRAME WINDOWS

Frame is used to create child windows within applets and top level or child windows for applications. It creates a standard style window. **Frame** supports the following constructors :

Frame()

Frame(String *title*)

The first form creates a window which does not contain a title while the second form creates a window with the title specified by *title*. Remember that you cannot specify the dimensions for the window, you must set the size of the window after it has been created. Some of the methods we shall use when working with **Frame** windows include:

The **setSize()** method is used to set the dimensions of the window. It takes the following form :

void setSize(int *newWidth*, int *newHeight*)

void setSize(Dimension *newSize*)

newWidth and *newSize* specify the size of the window in the first case. In the second case the size of the window is specified by the width and height fields of the **Dimension** object passed in *newSize*. The dimensions are specified in pixels.

To obtain the current size of the window the method is **getSize()**. Its form is:

Dimension getSize()

It returns the current size of the window in contained in the width and height fields of the **Dimension** object.

A frame window created will not be visible until you call the **setVisible()** method. This method is :

void setVisible(boolean *visibleFlag*)

If *visibleFlag* is **true**, the component is visible else it is hidden.

To change the title in the frame window use the method **setTitle()**. Its form is:

void setTitle(String *newTitle*)

where *newTitle* is the new title for the window.

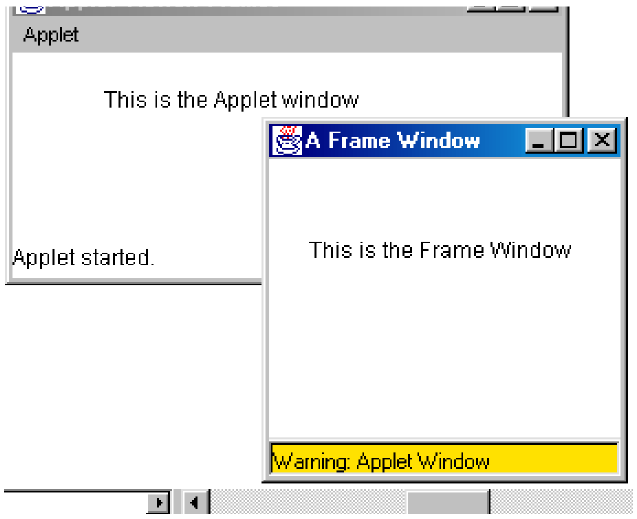
Your program must remove the window from the screen when it is closed by calling **setVisible**(**false**). To intercept a window close event, you should implement the **windowClosing()** method of the **WindowListener** interface.

Inside **windowClosing()** you must remove the window from the screen.

Let us now create a **Frame** window in an Applet as shown below:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code = “Frames” width = 300 height = 100>
</applet>
*/
class DemoFrame extends Frame {
        DemoFrame(String title) {
                super(title);
        }
public void paint(Graphics g) {
        g.drawString(“This is the Frame Window”, 50, 50);
}
public class Frames extends Applet {
public void init() {
        Frame f;
        f = new DemoFrame(“A Frame Window”);
        f.setSize(200,200);
        f.setVisible(true);
}
public void start() {
        f.setVisible(true);
}
public void stop() {
        f.setVisible(false);
}
public void paint(Graphics g) {
        g.drawString(“This is the Applet window”, 10, 30);
}
}
```

The sample output will be :



First you create a subclass of **Frame.** Once a subclass of **Frame** is created, you create an object of that class. This causes the frame window to come into existence, but you should call **setVisible()** (by setting it to true) to make it visible. You can also set the size of the window explicitly by calling **setSize()**. Next override any of the standard window methods like **init()**, **start()**, **stop()** and **paint()**. Note that we have not implemented any event handling mechanism. You can modify the above program to create an object to handle window events to close the frame window by implementing the **windowClosing()** method of the **WindowListener** interface by calling **setVisible()** .

12.3 Check Your Progress.

1. Fill in the blanks.

a) The ___________ method is used to set the dimensions of the Frame window.

b) A frame window created will not be visible until you call the ___________ method.

12.4 WORKING WITH GRAPHICS

12.4.1 Introduction

The AWT support a rich set of graphics methods. All graphics are drawn relative to a window. The window can be the main window of an applet, a child window of an applet or a stand alone application window. Each window originates at the upper left hand corner and is 0,0. All the output to a window takes place through a graphics context. A graphics context is encapsulated by the **Graphics** class and is obtained in the following ways :

- It is passed to an applet when one of its various methods like **paint()** or **update()** is called.

- It is returned by the **getGraphics()** method of **Component**.

The **Graphics** class defines a number of drawing functions. Each shape can be drawn edge only or filled. Objects are drawn and filled in the currently selected graphics color, which is black by default. Whenever a graphics object is drawn which exceeds the dimensions of the window, the output is automatically clipped. Let us see some of the drawing methods in this section.

12.4.2 Drawing Lines

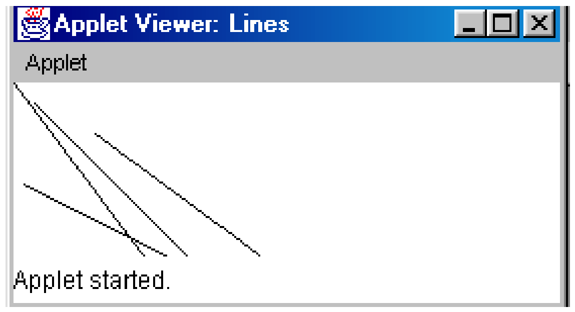
The method **drawLine()** is used to draw lines. Its form is :

void drawLine(int *startX*, int *startY*, int *endX*, int *endY*)

The line is drawn in the current drawing color beginning at *startX*, *startY* and ending at *endX*, *endY*. In the following example a number of lines are drawn :

```
import java.awt.*;
import java.applet.*;
/*
<applet code = "Lines" width = 300 height = 200>
</applet>
*/
public class Lines extends Applet {
      public void paint(Graphics g) {
               g.drawLine(10, 10, 100, 100);
               g.drawLine(0, 0, 150, 200);
               g.drawLine(5,50, 400, 250);
               g.drawLine(40, 25, 250, 180);
      }
}
```

The output of the program would be :



12.4.3 Drawing Rectangles :

drawRect() and **fillRect()** are the methods which display an outlined and filled rectangle respectively. These methods are :

void drawRect(int *top*, int *left*, int *width*, int *height*)

void fillRect(int *top*, int *left*, int *width*, int *height*)

top,*left* is the upper left corner of the rectangle. *width* and *height* specify the dimensions of the rectangle.

drawRoundRect() or **fillRoundRect()** are the methods to draw rounded rectangle. A rounded rectangle has rounded corners. The methods are:

void drawRoundRect(int *top*, int *left*, int *width*, int *height*, int *xDiam*, int *yDiam*)

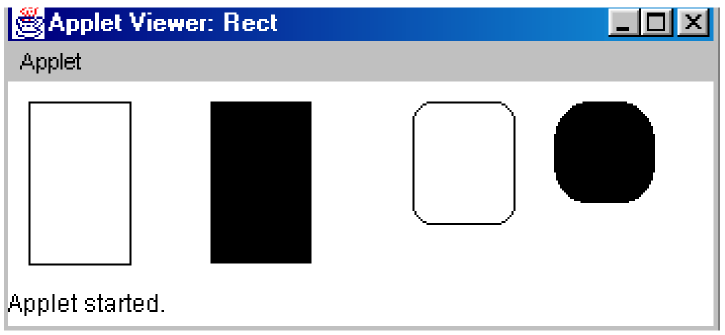
void fillRoundRect(int *top*, int *left*, int *width*, int *height*, int *xDiam*, int *yDiam*)

top,*left* is the upper left corner of the rectangle. *width* and *height* specify the dimensions of the rectangle. The diameter of the rounding arc along the X axis is specified by *xDiam* and the diameter of the rounding arc along the Y axis is specified by *yDiam*.

The following example illustrates

```
import java.awt.*;
import java.applet.*;
/*
<applet code = “Rect” width = 300 height = 200>
</applet>
*/
public class Rect extends Applet {
    public void paint(Graphics g) {
        g.drawRect(10, 10, 50, 80);
        g.fillRect(100, 10, 50, 80);
        g.drawRoundRect(200, 10, 50, 60, 20, 20);
        g.fillRoundRect(270, 10, 50, 50, 35, 40);
    }
}
```

output



12.4.4 Drawing Ellipses and Circles :

To draw an ellipse use the **drawOval()** method and to fill an ellipse use the **fillOval()** method. These methods are :

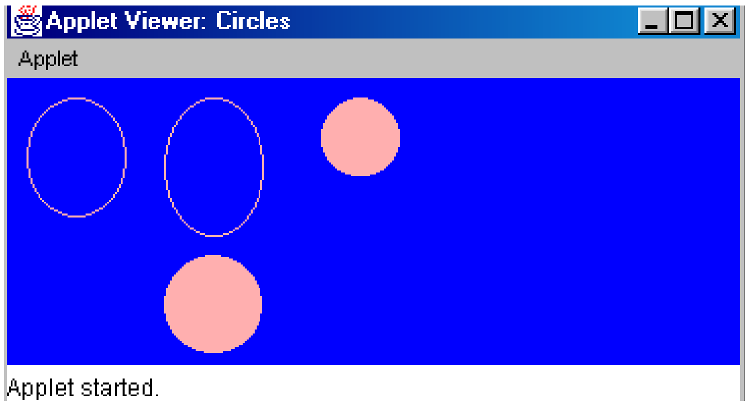
void drawOval(int *top*, int *left*, int *width*, int *height*)

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

The ellipse is drawn within a bounding rectangle whose upper left corner is *top*, *left* and whose width and height are specified by *width* and *height* respectively. To draw a circle you will have the specify a square as the bounding rectangle. The following example draws ellipses :

```
import java.awt.*;
import java.applet.*
/*
<applet code = “Circles” width = 300 height = 200>
</applet>
*/
public class Circles extends Applets {
        public void paint(Graphics g) {
                g.drawOval(10, 10, 50, 60);
                g.drawOval(80, 10, 50, 70);
                g.fillOval(160, 10, 40,40);
                g.fillOval(80, 90, 50, 50);
        }
}
```

output



***Note :** Many times you will want to size a graphics object to fit the current size of the window in which it is drawn. For this purpose, you should first obtain the current dimensions of the window by calling the **getSize()** method on the window object. It will return the dimensions of the window encapsulated within the*

***Dimension** object. Once you obtain the size of the window, then you can scale your graphical output as per requirement.*

12.4 Check Your Progress.

1. Match the following.

Column A	**Column B**
a) fillOval()	i) to draw a rounded rectangle
b) drawRect()	ii) to fill an ellipse
c) fillRect()	iii) to draw a rectangle
d) drawRoundRect()	iv) to fill a rectangle

12.5 COLOR

Java supports color in a portable, device independant fashion. You can specify any color that you want in the AWT. It then finds the best match for the color within the limits of the display hardware which is currently executing your program or applet. Color is encapsulated by the class **Color**. **Color** defines a number of constants to specify a number of common colors. You can also create your own colors by using the color constructors. Some of the constructors for **Color** are :

Color(int *red*, int *green*, int *blue*)

Color(int *rgbValue*)

Color(float *red*, float *green*, float *blue*)

Using the first constructor you can specify three integer values to specify the color as a mix of red, green and blue. These values must be between 0 and 255. eg.

new Color(100, 100, 255)

The second constructor takes a single integer value that contains a mix of red, green and blue packed in an integer. This integer is organised as : red in bits 16 to 23, green in bits 8 to 15 and blue in bits 0 to 7.

The third constructor takes three float values between 0.0 to 1.0 to specify the relative mix of red, green and blue.

The **setForeground()** and **setBackground()** are the methods which can be used to set the foreground and background colors respectively. You can also use the created color as the current drawing color.

You can obtain the red, green and blue components of a color independently using the following methods :

int getRed()

int getGreen()

int getBlue()

Each of these methods returns the RGB color component found in the invoking **Color** object in the lower 8 bits of an integer.

To obtain a packed RGB representation use the method **getRGB()**. Its form is :

int getRGB()

By default graphics objects are drawn in the current foreground color. You can change this color with the method **setColor()**. This method has the following form :

void setColor(Color *newColor*)

where *newColor* specifies the new drawing color.

getColor() is the method used to obtain the current color. Its form is :

Color getColor()

To make use of these methods to set colors and draw various objects like lines, rectangles, circles is left as an exercise to the student.

Setting the Paint Mode

The paint mode determines how objects are drawn in a window. By default the new output to a window overwrites any pre existing contents. However, it is possible to have new objects XORed onto the window. For this purpose you use the **setXORMode()** method which is as folows :

void setXORMode(Color *xorColor*)

where *xorColor* specifies the color that will be XORed to the window when an object is drawn. The XOR mode guarantees that the new object is always to be visible no matter what color the object is drawn over.

To return to the overwrite mode you use the **setPaintMode()** method. Its form is :

void setPaintMode()

In general, you will want to use the overwrite mode for normal output and the XOR mode for any special purposes.

12.5 Check Your Progress.

1. Fill in the blanks.

a) To obtain a packed RGB representation of the color we use the method_________

b) The __________ mode determines how objects are drawn in a window.

c) By default graphics objects are drawn in the current __________ color.

12.6 FONTS

The AWT supports multiple type fonts. Fonts have a family name, a logical font name and a face name. The family name is the general name of the font eg. Courier. The logical name specifies a catagory of the font eg. Monospaced. The face name specifies a specific font eg. Courier Italic. In this section let us see a few methods of the **Font** class and use these methods in examples to set fonts, obtain font information etc. Note that only an introduction to the **Font** class is presented here. A detailed study is an area the student would like to explore further.

Fonts are encapsulated in the **Font** class which defines several methods. It also defines the following variables :

Variable	**Meaning**
String name	Name of the font
float pointSize	Size of the font in points
int size	Size of the font in points
int style	Font style

In this section we shall attempt an overview of the methods of the **Font** class. A detailed study of these methods may be an area the student may want to explore on his own.

To obtain information regarding the available fonts on your machine you use the **getAvailableFontFamilyNames()** method. This method is defined by the **GraphicsEnvironment** class. This method is shown below :

```
String[ ] getAvailableFontFamilyNames()
```

This method returns an array of strings that contains the names of the available font families.

The **getAllFonts()** method returns an array of **Font** objects for all the available fonts. This method is defined by the **GraphicsEnvironment** class and has the following form :

```
Font[ ] getAllFonts()
```

Both these methods are members of **GraphicsEnvironment**, so you need a **GraphicsEnvironment** reference to call them. This reference is obtained by the **getLocalGraphicsEnvironment() static** method as :

```
staticGraphicsEnvironment getLocalGraphicsEnvironment()
```

The following example illustrates how to obtain the names of the available fonts :

```
import java.applet.*;
import java.awt.*;
/*
<applet code = “Fonts” width = 400 height = 200>
</applet>
*/
public class Fonts extends Applet {
                        public void paint(Graphics g) {
                              String s = “ “;
```

```
                    String FList[ ];
                    GraphicsEnvironment ge =
GraphicsEnvironment.getLocalGraphicsEnvironment();
                    FList = ge.getAvailableFontFamilyNames();
                    for(int i = 0; i < FList.length; i++)
                          s+= FList[i] + " ";
                    g.drawString(s, 10, 10);
              }
          }
```

Run the program and check its output.

To select a new font, you should first construct a **Font** object that describes the font. One of the constructors of **Font** which is used for this purpose is :

Font(String *fontName*, int *fontStyle*, int *pointSize*)

fontName specifies the name of the desired font, which can be specified either using the logical name or the face name. The default font, if you don't explicitly set a font is the **Dialog** font. This is also the font used by your system's Dialog boxes. The style of the font is specified by *fontStyle*. It may consist of one or more of the three constants : **Font.PLAIN**, **Font.BOLD**, **Font.ITALIC**. To combine the styles you have to OR them together eg. Font.BOLD | Font.ITALIC specifies a bold, italic style. *pointSize* is used to specify the size of the font in points.

In order to use the font that you have created you should select it using the **setFont()** method which is defined by **Component**. Its general form is :

void setFont(Font *fontObj*)

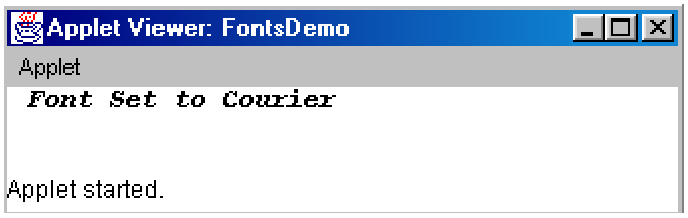
where *fontObj* is the object that contains the desired font.

The following program illustrates the use of the methods discussed above.

```
import java.awt.*;
import java.applet.*;
/*
<applet code = "FontsDemo" width = 200 height = 200>
</applet>
*/
public class FontsDemo extends Applet {
      Font f;
      String s;
      public void init() {
```

```
                    f = new Font("Courier", Font.BOLD |
                    Font.ITALIC, 14);
                    setFont(f);
                    s = "Font Set to Courier";
                }
                public void paint(Graphics g) {
                    g.drawString(s, 10, 10);
                }
}
```

An output of the program :



```
Font Set to Courier
```

```
Applet started.
```

To obtain information about the currently selected font, you should first get the current font by calling the method **getFont()**. This method is defined by the **Graphics** class. It has the following form :

Font getFont()

Once you have obtained the currently selected font, you can retrieve information about the font, using various methods defined by **Font** as illustrated in the example below:

```
import java.applet.*;
import java.awt.*;
/*
<applet code = "FontInfo" width = 300 height = 100>
</applet>
*/
public class FontInfo extends Applet {
                public void paint(Graphics g) {
                    Font f = g.getFont();
                    String name = f.getName();
                    String family = f.getFamily();
                    int size = f.getSize();
                    int style = f.getStyle();
                    String s = "Font Family : " + family;
```

```
              s += “  Font : “ + name; + “  Size : “ + size + “
         Style : “;
              if((style & Font.BOLD) == Font.BOLD)
                   s+= “Bold”;
              if((style & Font.ITALIC) == Font.ITALIC)
                   s+= “Italic”;
              if((style & Font.PLAIN) == Font.PLAIN)
                   s+= “Plain”;
              g.drawString(s, 20, 20);
         }
    }
```

A sample output of the above program is :

12.7 SUMMARY

The AWT (Abstract Window Toolkit) contains numerous classes and methods which allow you to create and manage windows. The AWT classes are contained in the java.awt package. It is one of the largest packages of Java. The AWT defines windows according to a class hierarchy that adds functionality and specificity with each level. Those are Panel, Frame, Component, Container, Canvas.

12.8 CHECK YOUR PROGRESS - ANSWERS

12.1 & 12.2

1. a) java.awt

b) Panel, Frame

c) container

2. a) False

b) False

12.3

1. a) setSize()

b) setVisible()

12.4

1. a) - (ii)

b) - (iii)

c) - (iv)

d) - (i)

12.5

1. a) getRGB()

b) paint

c) foreground

12.6

1. a) Font

b) Dialog

12.9 QUESTIONS FOR SELF - STUDY

1. What is the use of a Frame? Describe some methods of Frame.
2. What are the ways in which you can obtain a Graphics context? Describe a few drawing methods of the Graphics class.
3. Describe the constructors of Color.
4. Describe the methods of the Font class. Also list the variables defined in the Font class.
5. **Write short notes on :**

   a) Component Class

   b) Panel Class

   c) Setting the paint mode.

12.10 SUGGESTED READINGS

1. www.java.com
2. www.freejavaguide.com
3. www.java-made-easy.com
4. www.tutorialspoint.com
5. www.roseindia.net

❑ ❑ ❑

Notes

Chapter : 13

AWT Controls and Layout Managers

13.0 OBJECTIVES

Dear students,

After studying this chapter you will able to :

- explain the control components
- discuss how to use the controls like Labels, Push buttons, Check boxes, Check lists, Lists, Scroll bars, and Text editing.
- discuss the types of layouts viz. flow layout, border layout, grid layout, card layout and their use in programs.

13.1 INTRODUCTION

Controls are the components that allow the user to interact with your application. eg. a push button control. A layout manager automatically positions controls within a container. In addition to controls, a frame window can also include a menu bar. Each entry in a menu bar activates a drop down menu of options. A user can select the option of his choice. A menu bar is always positioned at the top of a window.

The controls supported by AWT are subclasses of **Component** and include the following : Labels, Push buttons, Check boxes, Check lists, Lists, Scroll bars, and Text editing.

To include a control in the window, it has to be added to the window. For this purpose, you should create an instance of the control and add it by using the method **add()**. This method is defined by **Container**. Its form which we shall use for controls is:

Component add(Component *compObj*)

where *compObj* is an instance of the control that you want to add. Once a control has been added, it is automatically visible whenever its parent window is displayed.

To remove a control from the window we use the method **remove()** which is also defined by **Component**. Its general form is :

void remove(Component *obj*)

where *obj* is the reference to the control you want to remove. You can also remove all the controls by calling the method **removeAll()**.

Labels are passive controls whereas all the other controls generate events when they are accessed by the user. Your program therefore should implement the appropriate interface and register an event listener for each control which is to be monitored. Once you install a listener, the event will automatically be sent to it.

13.1 Check Your Progress.

1. Write True or False.

a) Check Boxes are passive controls.

b) Labels generate events when they are accessed by the user.

c) A layout manager automatically positions controls within a container.

13.2 AWT CONTROLS

Let us study the various controls supported by AWT in the following sections:

13.2.1 Labels

A label is an object of type **Label** and contains a string which it displays. Labels do not support any interaction with the user and are therefore passive controls. The constructors for Label are :

Label()

Label(String *str*)

Label(String *str*, int *how*)

where the string is specified by *str*. This string is by default left justified. *how* specifies the alignment which will be used by *str*. The value of how must be one of these three constants : **Label.LEFT**, **Label.RIGHT**, **Label.CENTER**.

setText() method is used to set or change text in a label. The **getText()** method is used to obtain the current label. These methods have the following form :

void setText(String *str*)	*str* sets the string for the label
String getText()	returns the current label

To set the alignment of the string within the label you use the method :

void setAlignment(int *how*)

how is the alignment constant already discussed.

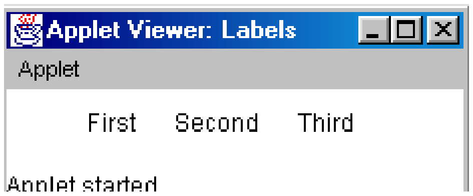
getAlignment() is used to obtain the current alignment :

int getAlignment()

The following example will demonstrate :

```
import java.awt.*;
import java.applet.*;
/*
<applet code = "Labels" width = 200 height = 300>
</applet>
*/
public class Labels extends Applet {
    public void init() {
            Label first = new Label("First");
            Label second = new Label("Second");    // Create Labels
            Label third = new Label("Third");
            add(first);
            add(second);                         // Add labels to applet window
            add(third);
    }
}
```

output



13.2.2 Buttons

A push button is a component that contains a label and generates an event when pressed. Push buttons are objects of type **Button**. **Button** defines the following constructors :

Button()   Button(String *str*)

The second constructor creates a button which contains *str* as a label.

After a button has been created, you can set its label with the **setLabel()** method. The label can be retrieved using the **getLabel()** method. These methods are :

void setLabel(String *str*)

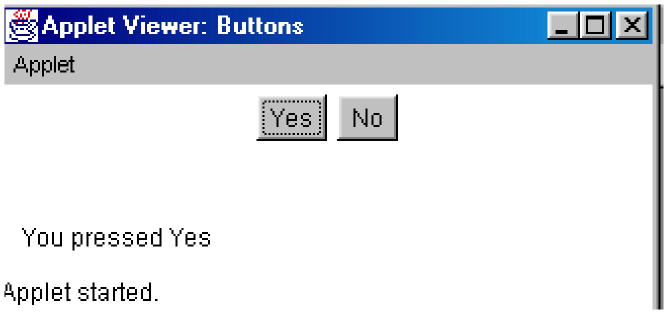
String getLabel()

Each time a button is pressed an action event is generated which is sent to any listeners which have previously registered for receiving the action event notification. Each listener implements the **ActionListener** interface. This interface defines the **actionPerformed()** method. This method is called when an event occurs. It takes an **ActionEvent** object as its argument. This object contains both a reference to the button that generated the event and a reference to the string that is the label of the button. Either value may be used to identify the button.

The following example illustrates using the label of the button to determine which button is pressed. Each time a button is pressed, a message is displayed which reports which button is displayed. Note that we obtain the label by calling the **getActionCommand()** method on the **ActionEvent** object passed to **actionPerformed()**.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code = “Buttons” width = 100 height = 300>
</applet>
*/
public class Buttons extends Applet implements ActionListener {
        String s = “ “;
        Button yes, no;
        public void init() {
                yes = new Button(“Yes”);
                no = new Button(“No”);
                add(yes);
                add(no);
                yes.addActionListener(this);
                no.addActionListener(this);
        }
        public void actionPerformed(ActionEvent ae) {
                String s1 = ae.getActionCommand();
                if(s1.equals(“Yes”))
                        s = “You pressed Yes”;
                else
                        s = “You pressed No”;
                repaint();
        }
        public void paint(Graphics g) {
                g.drawString(s, 10, 80);
        }
}
```

A sample output of the above program is as shown below:



```
You pressed Yes
Applet started.
```

The following example shows how to use the **getSource()** method to obtain the object to determine which button has been pressed. For this purpose you should keep a list of the button objects when they are created.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code = “Buttons1” width = 250 height = 100>
</applet>
*/
public class Buttons1 extends Applet implements ActionListener {
        String s= “ “;
        Button bList[] = new Button[2];
        public void init() {
                Button yes = new Button("Yes");
                Button no = new Button("No");
                bList[0] = (Button) add(yes);
                bList[1] = (Button) add(no);
                for(int i = 0; i < 2; i++)
                        bList[i].addActionListener(this);
        }
        public void actionPerformed(ActionEvent ae) {
                for(int i = 0; i < 2; i++)
                {
                        if(ae.getSource() == bList[i])
                                s = "You pressed " + bList[i].getLabel();
                }
                repaint();
        }
        public void paint(Graphics g) {
                g.drawString(s, 10, 80);
        }
}
```

Check the output of this program as an exercise.

13.2.1 & 13.2.2 Check Your Progress.

1. Fill in the blanks.

a) .... method is used to set or change text in alabel.

b) Push buttons are objects of type ....

c) The .... method is used to obtain the object to determine which button has been pressed.

13.2.3 Check Boxes

A check box is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not. There is a label associated with each check box. It describes the option that the check box represents. Check boxes can be used indivisually or as part of a group. Check boxes are objects of the **Checkbox** class. Its constructors are :

Checkbox()   Checkbox(String *str*)

Checkbox(String *str*, boolean *on*)

Checkbox(String *str*, boolean *on*, CheckboxGroup *cbGroup*)

Checkbox(String *str*, CheckboxGroup *cbGroup*, boolean *on*)

str specifies the label for a check box. *on* allows you to set the initial state of the check box. If *on* is **true** then the checkbox is initially checked, otherwise it is cleared. *cbGroup* specifies the group for a checkbox. If the check box is not a part of the group then *cbGroup* must be **null**.

getState() method used to retrieve the current state of a check box. Its form is :

boolean getState()

setState() is used to set the state of the check box. Its form is :

void setState(Boolean *on*)

If *on* is **true**, the box is checked, if **false** the box is cleared.

To obtain the current label, you use **getLabel()** and to set the label you use **setLabel()**. These methods are as follows :

String getLabel()   void setLabel(String *str*)

where *str* specifies the label for the check box.

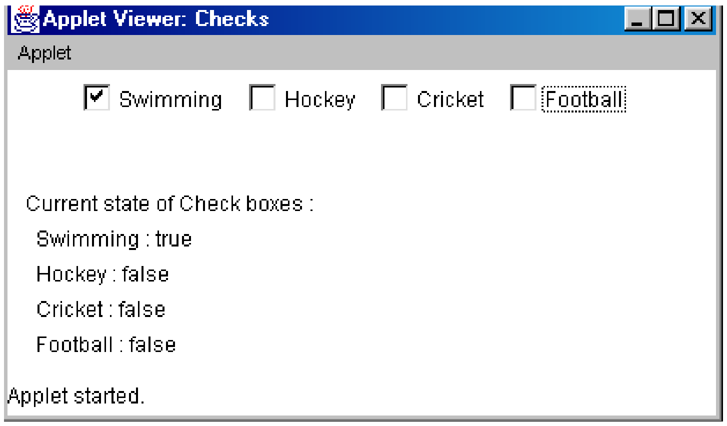
Each time a check box is checked or cleared, an item event is generated which is sent to any listener which previously registered an interest in receiving item event notifications from that component. Each listener implements the **ItemListener** interface. This interface defines the **itemStateChanged()** method which takes as its argument an object of **ItemEvent**.

In the following example we illustrate the use of check boxes and display status of each box. Every time you change the status of the check box, the status display will be updated.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code = “Checks” width = 240 height = 200>
</applet>
*/
public class Checks extends Applet implements ItemListener {
    String s = “ “;
    Checkbox swim, hockey, cricket, football;
    public void init() {
            swim = new Checkbox(“Swimming”, true);
            hockey = new Checkbox(“Hockey”);
            cricket = new Checkbox(“Cricket”);
            football = new Checkbox(“Football”);
```

```
                add(swim);
                add(hockey);
                add(cricket);
                add(football);
                swim.addItemListener(this);
                hockey.addItemListener(this);
                cricket.addItemListener(this);
                football.addItemListener(this);
        }
        public void itemStateChanged(ItemEvent ie) {
                repaint();
        }
        public void paint(Graphics g) {
                s = "Current state of Check boxes :";
                g.drawString(s, 10, 80);
                s = "  Swimming : " + swim.getState();
                g.drawString(s, 10, 100);
                s = "  Hockey : " + hockey.getState();
                g.drawString(s, 10, 120);
                s = "  Cricket : " + cricket.getState();
                g.drawString(s, 10, 140);
                s = "  Football : " + football.getState();
                g.drawString(s, 10, 160);
        }
}
```

output



```
Current state of Check boxes :
 Swimming : true
 Hockey : false
 Cricket : false
 Football : false
Applet started.
```

13.2.4 Checkbox Group

Radio buttons are a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time. In order to create radio buttons you must first define a group to which they belong and then specify that group when you construct the check boxes. Check box groups are objects of type **CheckBoxGroup**. It has only the default constructor defined, which creates an empty group.

To determine which check box in a group is currently selected you call the **getSelectedCheckbox()** method. To set a check box you call the **setSelectedCheckbox()** method. These methods are as follows :

```
Checkbox getSelectedCheckbox()
void setSelectedCheckbox(Checkbox which)
```

where *which* is the check box you want to be selected. In this case, the previously selected check box will be turned off.

The following example will demonstrate :

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code = “CheckGroup” width = 300 height = 200>
</applet>
*/
public class CheckGroup extends Applet implements ItemListener {
    String s=   “ “;
    Checkbox swim, hockey, cricket, football;
    CheckboxGroup cbg;
    public void init() {
            cbg = new CheckboxGroup();
            swim = new Checkbox(“Swimming”, cbg, true);
            hockey = new Checkbox(“Hockey”, cbg, false);
            cricket = new Checkbox(“Cricket”, cbg, false);
            football = new Checkbox(“Football”, cbg, false);
            add(swim);
            add(hockey);
            add(cricket);
            add(football);
            swim.addItemListener(this);
            hockey.addItemListener(this);
```

```
                cricket.addItemListener(this);
                football.addItemListener(this);
        }
        public void itemStateChanged(ItemEvent ie) {
                repaint();
        }
        public void paint(Graphics g) {
                s = “Currently selected : “;
                s+=cbg.getSelectedCheckbox().getLabel();
                g.drawString(s, 10, 100);
        }
}
```

Run the program and check output as an exercise.

13.2.3 & 13.2.4 Check Your Progress.

1. Write True or False.

a) A check box is a control that is used to turn an option on or off.

b) More than one Radio buttons can be checked at a time in a checkbox group.

c) **CheckBoxGroup** has only the constructor defined, which creates an empty group.

13.2.5 Choice Controls

The **Choice** class is used to create pop up list of items from which the user can select. Thus, a *Choice control* is a form of menu. When the user clicks on a **Choice** component, the whole list of choices pops up and a new selection can be made. Each item in the list appears in the order in which it has been added to the **Choice** object. **Choice** defines only the default constructor which creates an empty list.

To add a selection to the list call **addItem()** or **add()**. The general forms of these methods are :

void addItem(String *name*)

void add(String *name*)

where *name* is the name of the item being added. To determine which item is currently selected you can call either the **getSelectedItem()** or the **getSelectedIndex()** methods. These methods are :

String getSelectedItem()

This method returns a string containing the name of the item

int getSelectedIndex()

returns the index of the item. The first item is index 0.

By default the first item added to the list is selected.

With **getItemCount()** method you can obtain the number of items in the list.

Its form is :   int getItemCount()

You can set the currently selected item using the **select()** method with either a zero based integer index or a string that will match a name in the list. These methods are:

void select(int *index*)   voidselect(String *name*)

You can obtain the name associated with a particular item with the **getItem()** method. Its general form is :

String getItem(int *index*)

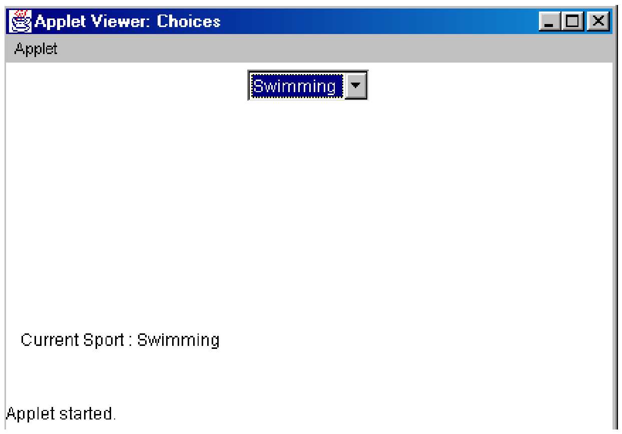
where *index* specifies the index of the desired item.

Everytime a choice is selected, an item event is generated. This is sent to any previously registered listener. Each listener implements the **ItemListener** interface. This interface defines the **itemStateChanged()** method, which takes an **ItemEvent** object as its argument.

The following program demonstrates creating choice menus:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code = “Choices” width = 250 height = 300>
</applet>
*/
public class Choices extends Applets implements ItemListener {
    Choice sports;
    String s = “ “;
    public void init() {
            sports = new Choice();
            sports.add(“Swimming”);
            sports.add(“Hockey”);
            sports.add(“Cricket”);
            sports.add(“Football”);
            add(sports);
            sports.addItemListener(this);
    }
    public void itemStateChanged(ItemEvent ie) {
```

```
                repaint();
            }
            public void paint(Graphics g) {
                s = "Current Sport : ";
                s += sports.getSelectedItem();
                g.drawString(s, 10, 200);
            }
    }
```
output



Current Sport : Swimming

Applet started.

13.2.6 Lists

The **List** class provides a compact multiple-choice scrolling selection list. In the **Choice** object, only the single selected item in the menu is displayed. However, a **List** object can be constructed to show any number of choices in the visible window. It can also be created to allow multiple selections.

Constructors for **List** are :

List()

List(int *numRows*)

List(int *numRows*, boolean *multipleSelect*)

where *numRows* specifies the number of entries in the list that will always be visible. Others are scrolled into view as needed. If *multipleSelect* is **true**, then the user may select two or more items at a time, if it is **false**, then only one item can be selected.

Use the method **add()** to add a selection to the list.

It has the following two forms :

void add(String *name*)   void add(String *name*, int *index*)

where *name* is the name of the item to be added to the list. In the second form of the method, you can specify the index at which you wish to add the item to the list by *index*. Indexing begins at 0.

In lists which allow only single selection you can determine the currently selected item with either the **getSelectedItem()** or **getSelectedIndex()** methods.

These methods are :

String getSelectedItem()   int getSelectedIndex()

getSelectedItem() returns a string containing the name of the item. A **null** is returned is no item is selected or if more than one item is selected. The **getSelectedIndex()** method returns the index of the item. Remember that the first item is at index 0. If more than one item is selected or no selection has been done a -1 is returned.

For lists which allow multiple selection, the methods which can be used to determine the selected items are :

String[ ] getSelectedItems()   int[ ] getSelectedIndexes()

getSelectedItems() returns an array containing the names of the currently selected items. **getSelectedIndexes()** returns an array containing the indexes of the currently selected items.

The **getItemCount()** method is used to obtain the number of items in the list. Its form is :

int getItemCount()

You can set the currently selected item by using the **select()** method with a zero based integer index. Its form is :

void select(int *index*)

You can obtain the name associated with an item at a particular index with the **getItem()** method. Its general form is :

String getItem(int *index*)

where *index* specifies the index of the desired item.

Each time a **List** item is double clicked an **ActionEvent** object is generated. Its **getActionCommand()** method can be used to retrieve the name of the newly selected item. Also, everytime an item is selected or deselected with a single click, an **ItemEvent** object is generated. Its **getStateChanged()** method can be used to determine whether a selection or deselection triggered the event. **getItemSelectable()** returns a reference to the object that triggered the event.

The following example illustrates two **List** components, one with single choice and the other with multiple choice.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code= “Lists” width = 250 height = 300>
</applet>
```

```
*/
public class Lists extends Applet implements ActionListener {
        List sports, hobbies;
        String s = " ";
        public void init() {
                sports = new List(4, true);
                hobbies = new List(5, false);
                sports.add("Swimming");
                sports.add("Hockey");
                sports.add("Cricket");
                sports.add("Football");
                hobbies.add("Reading");
                hobbies.add("Painting");
                hobbies.add("Cooking");
                hobbies.add("Drawing");
                hobbies.add("StampCollecting");
                add(sports);
                add(hobbies);
                sports.addActionListener(this);
                hobbies.addActionListener(this);
        }
        public void actionPerformed(ActionEvent ae) {
                repaint();
        }
        public void paint(Graphics g) {
                int i[ ];
                s = "Sports Selected : ";
                i = sports.getSelectedIndexes();
                for(int j = 0; j < i.length; j++)
                        s+= sports.getItem(i[j]) + " ";
                g.drawString(s, 10, 200);
                s = "Hobby selected :  ";
                s += hobbies.getSelectedItem();
                g.drawString(s, 10, 220);
        }
}
```

To run the program and check the results is left as an exercise to the student.

13.2.5 & 13.2.6 Check Your Progress.

1. Write True or False.

a) A Choice control is a form of menu. .

b) Everytime a choice is selected, an item event is generated.

c) **List** object cannot be created to allow multiple selections.

d) **getSelectedItem()** returns a **null** if more than one item is selected.

13.2.7 Text Field

The **TextField** class implements a single line text entry area usually called as *edit control*. Text fields allow the user to enter strings and to edit the text using arrow keys, cut and paste keys and mouse selections. **TextField** is a subclass of **TextComponent**. It defines the following constructors :

TextField() TextField(int *numChars*)

TextField(String *str*) TextField(String *str*, int *numChars*)

The first form creates a default text field. The second form creates a text field which is *numChars* wide. The third form initialises the text field with the string contained in *str*, whereas in the fourth form you can initialise the text field and set its width.

To obtain the string currently contained in the text field you call the **getText()** method and to set the text you call the **setText()** method. These methods are :

String getText()

void setText(String *str*)

where str is the new string.

You can select a portion of the text in the text field and you can also select a portion of the text under program control. Both these can be achieved with the **select()** method. The currently selected text can be obtained by your program by using **getSelectedText()**. These methods are :

String getSelectedText()

void select(int *startIndex*, int *endIndex*)

The **select()** method selects characters beginning at *startIndex* and ending at *endIndex*-1.

To control whether the contents of the text field may be modified by the user or not you use the **setEditable()** method. Its form is :

void setEditable(boolean *canEdit*)

Here, if *canEdit* is **true** then the text maybe changed, if it is **false** the text cannot be altered.

isEditable() is the method used to determine the editability of the text field. Its form is :

boolean isEditable()

it returns **true** if the text maybe changed, **false** otherwise.

You can disable the character echoing on the screen as they are typed by the **setEchoChar()** method. This method specifies a single character that the text field will display when characters are entered. This means that the actual characters typed will not be shown. This is particularly useful when entering passwords.

This method has the form :

void setEchoChar(char *ch*)

where *ch* specifies the character to be echoed.

You can retrieve the echo character by calling the **getEchoChar()** method.

Its form is :

char getEchoChar()

echoCharIsSet() is the method which can be used to check whether a text field is in this mode. Its form is :

boolean getCharIsSet()

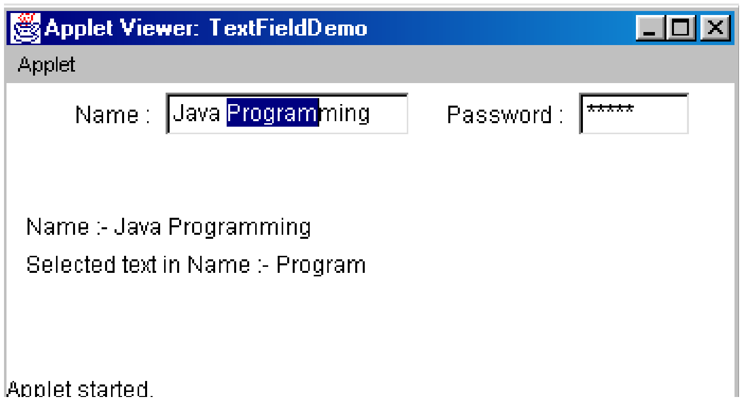
Text fields perform their own editing functions. So your program do not need to respond to indivisual key events that occur within the text field. However, when the user presses Enter, you may want to respond. This generates an action event.

Let us create the user name and password screen in the following example which will make use of many methods which have been discussed above :

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code = “TextFieldDemo” width = 300 height = 200>
</applet>
*/
public class TextFieldDemo extends Applet implements ActionListener {
        TextField name, pass;
        public void init() {
        Label n = new Label(“Name : “, Label.RIGHT);
        Label p = new Label(“Password : “, Label.RIGHT);
                name = new TextField(15);
                pass = new TextField(5);
                pass.setEchoChar(‘*’);

                add(n);
                add(name);
                add(p);
                add(pass);
                name.addActionListener(this);
                pass.addActionListener(this);
        }
        public void actionPerformed(ActionEvent ae) {
                repaint();
        }
        public void paint(Graphics g) {
          g.drawString("Name :- " + name.getText(), 10, 80);
                g.drawString("Selected text in Name :- " +
                        name.getSelectedText(), 10,100);
        }
}
```

A sample output of the program is as shown below :



```
Name :- Java Programming
Selected text in Name :- Program
```

```
Applet started.
```

13.2.8 Scroll Bars

Scroll bars are used to select continuous values between a specified minimum and maximum. Scroll bars can be oriented horizontally or vertically. A scroll bar is in fact a composite of a number of indivisual parts. Each end of the scroll bar has an arrow. You can click on these arrows to move the current value of the scroll bar by one unit in the direction of the arrow. The current value of the scroll bar relative to its minimum and maximum values is indicated by the slider box for the scroll bar. The slider box (or *thumb*) can be dragged by the user to a new position. The scroll bar then reflects this value. If the user clicks in the background space on either side of the thumb, then it causes the thumb to jump in that direction by an increment value larger than 1. Typically, this action translates into some form of page up or page down. The **Scrollbar** class encapsulates scroll bars. **Scrollbar** defines the following constructors :

Scrollbar()   Scrollbar(int *style*)

Scrollbar(int *style*, int *initialValue*, int *thumbSize*, int *min*, int *max*)

The first form creates a vertical scroll bar. With the second and third forms you can specify the orientation of the scroll bar. If style is **Scrollbar.VERTICAL** a vertical scroll bar is created. If style is **Scrollbar.HORIZONTAL** then a horizontal scrollbar is created. The third form allows you to pass the intial value of the scroll bar in *initialValue*. The number of units represented by the height of the thumb is passed in *thumbSize*. *min* and *max* specify the minimum and maximum values of the scroll bar respectively.

The **setValues()** method is used to set the parameters for the scroll bar if the first or the second form of constructor is used for constructing the scroll bar. The **setValues()** method has the form:

void setValues(int *initialValue*, int *thumbSize*, int *min*, int *max*)

Here, the parameters have the same meaning as they have in the third form of constructor described above.

getValue() is invoked to obtain the current value of the scroll bar. It returns the current setting. **setValue()** is called to set the current value. These methods are :

int getValue()

void setValue(int *newValue*)

newValue specifies the new value for the scroll bar. When you set value the slider box inside the scroll bar will be positioned to reflect the new value.

The minimum and maximum values can also be retrieved using the **getMinimum()** and **getMaximum()** methods which are:

int getMinimum()

int getMaximum()

They both return the requested value.

Each time a scroll bar is scrolled up or down one line, then by default 1 is the increment added to or subtracted from the scroll bar. This increment can be changed by calling the **setUnitIncrement()** method. page up and page down increments are 10 by default. This value can also be changed by calling the **setBlockIncrement()** method. These methods are :

void setUnitIncrement(int *newIncr*)

void setBlockIncrement(int *newIcr*)

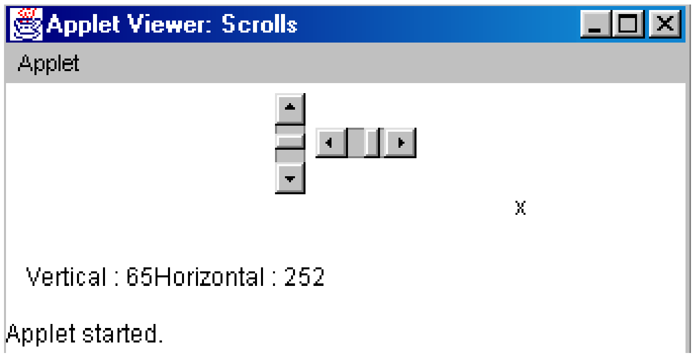
The **AdjustmentListener** interface is needed to process scroll bar events. Every time the user interacts with a scroll bar, an **AdjustmentEvent** object is generated. The **getAdjustmentType()** method is used to determine the type of the adjustment. Following are the adjustment types :

BLOCK_DECREMENT	A page-down event has been generated
BLOCK_INCREMENT	A page-up event has been generated
TRACK	An absolute tracking event has been generated
UNIT_DECREMENT	The line-down button in a scroll bar has been pressed
UNIT_INCREMENT	The line-up button in a scroll bar has been pressed

In the following example, we shall create both horizontal and vertical scroll bars and display the current settings of the scroll bar. The scroll bars will be updated if you drag the mouse while inside the window.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code = “Scrolls” width = 300 height = 150>
</applet>
*/
public class Scrolls extends Applet
        implements AdjustmentListener, MouseMotionListener {
                String s = “ “;
                Scrollbar vert, hor;
                public void init() {
                int width =
Integer.parseInt(getParameter(“width”));
                int height =
Integer.parseInt(getParameter(“height”));
                vert = new Scrollbar(Scrollbar.VERTICAL, 0, 1,
0, height);
                hor = new  Scrollbar(Scrollbar.HORZIONTAL, 0,
1, 0, width);
                add(vert);
                add(hor);
                vert.addAdjustmentListener(this);
                hor.addAdjustmentListener(this);
                addMouseMotionListener(this);
        }
        public void adjustmentValueChanged(AdjustmentEvent
ae) {
                repaint();
        }
        public void mouseDragged(MouseEvent me) {
                int x = me.getX();
                int y = me.getY();
                vert.setValue(y);
                hor.setValue(x);
                repaint();
        }
        public void mouseMoved(MouseEvent me) {
        }                       // implement mouseMoved
        public void paint(Graphics g) {
                s = "Vertical : "+ vert.getValue();
                s += "Horizontal : " + hor.getValue();
                g.drawString(s, 10, 100);
           g.drawString("x", hor.getValue(), vert.getValue());
                        // current mouse drag position
        }
    }
```

A sample output of the above program :



```
Vertical : 65Horizontal : 252
Applet started.
```

13.2.7 & 13.2.8 Check Your Progress.

1. **Fill in the blanks.**

a) **TextField** class implements a single line text entry area called as ....

b) **isEditable()** returns ....if the text in a textfield maybe changed.

c) .... are used to select continuous values between a specified minimum and maximum.

d) Each time a scroll bar is scrolled up one line, by default .... is the increment added to the scroll bar.

13.2.9 Text Area

The AWT includes a simple multline editor called **TextArea**. Its constructors are :

TextArea()   TextArea(int *numLines*, int *numChars*)

TextArea(String *str*)   TextArea(String *str*, int *numLines*, int *numChars*)

TextArea(String *str*, int *numLines*, int *numChars*, int *sBars*)

numLines specifies the height, in terms of lines of the text area and *numChars* specifies its width, in characters. Initial text can be specified in *str*. *sBars* allows to specify the scroll bars you want the control to have. It can have one of the following values:

SCROLLBARS_BOTH	SCROLLBARS_NONE
SCROLLBARS_HORIZONTAL_ONLY	SCROLLBARS_VERTICAL_ONLY

TextArea is a subclass of **TextComponent**. It supports the **getText()**, **setText()**, **getSelectedText()**, **select()**, **isEditable()**, and **setEditable()** methods which have been previously described.

TextArea adds the following methods :

void append(String *str*) : appends the string specified by *str* to the end of the current text.

void insert(String *str*, int *index*) : inserts the string specified by *str* at the specified index.

void replaceRange(String *str*, int *startIndex*, int *endIndex*) : This method replaces characters from *startIndex* to *endIndex*-1. The replacement text is passed in *str*.

Text areas are almost self contained controls.

The following program demonstrates the use of text areas :

```
import java.awt.*;
import java.applet.*;
/*
<applet code = “TextDemo” width = 400 height = 250>
</applet>
*/
public class TextDemo extends Applet {
        public void init() {
                String val = “Scroll bars are used to select
continuous values\n” +
                        “ between a specified minimum
        and maximum.\n” +
                        “ Scroll bars can be oriented
        horizontally or vertically.\n” +
                        “A scroll bar is in fact a composite
        of\n” +
                        “ a number of indivisual parts.\n “;
                TextArea text = new TextArea(val, 10, 30);
                add(text);
        }
}
```

Check output of the above program to study the application of **TextArea** control.

13.2.9 Check Your Progress.

1. Fill in the blanks.

a) **TextArea** is a .... editor.

b) **TextArea** is a subclass of ....

13.3 LAYOUT MANAGERS

13.3.1 Fundamentals

In all our previous examples, all the components have been positioned by the default layout manager. A layout manager automatically arranges the controls within a window. Each **Container** object has a layout manager associated with it. A layout manager is an instance of any class that implements the **LayoutManager** interface. The layout manager is set by the **setLayout()** method. If no call to **setLayout()** is made, then the default layout manager is used. Whenever a container is resized (or sized for the first time), the layout manager is used to position the components in it.

The **setLayout()** method has the following form :

void setLayout(LayoutManager *layoutObj*)

wher *layoutObj* is a reference to the desired layout manager. It is also possible to the disable the layout manager and position the components manually. For this purpose, you pass a **null** for *layoutObj*. In such a situation, you should determine the shape and position of each component manually by using the **setBounds()** method defined by **Component**. Normally layout manager is used.

Each layout manager keeps a track of the list of components which are stored by their names. The layout manager is notified everytime you add a component to a container. Each component that is being managed by the layout manager contains the **getPreferredSize()** and **getMinimunSize()** methods. These return the preferred size and the minimum size required for each component. Whenever the container needs to be resized, the **minimumLayoutSize()** and **preferredLayoutSize()** methods are used. In this section let us study the predefined **LayoutManager** classes provided by Java.

13.3.2 Flow Layout

This is the default layout manager. This was the layout manager which was used by all our preceding examples. In the **FlowLayout**, components are laid out from the upper left hand corner, left to right and top to bottom. When no more components fit on a line, the next component appears on the next line. A small space is left between each component, above and below it as well as to the left and right.

The constructors for **FlowLayout** :

FlowLayout()

FlowLayout(int *how*)   FlowLayout(int *how*, int *horz*, int *vert*)

In the first form, all components are centered and a space of 5 pixels is left between each component. In the second form, you can sepcify how each line is aligned. The valid values for *how* are :

FlowLayout.LEFT   FlowLayout.CENTER   FlowLayout.RIGHT

These values specify the left, center and right alignment respectively. The third form allows to specify the horizontal and vertical space left between components in *horz* and *vert* respectively.

Let us modify our previous example of **CheckBox** control so that it uses the right aligned flow layout. Note that this program is for the purpose of illustration of the layout manager. No event handling mechanisms are incorporated.

```
import java.awt.*;
import java.applet.*;
/*
<applet code = “CheckGroup” width = 300 height = 200>
</applet>
```

```
*/
public class CheckGroup extends Applet {
        Checkbox swim, hockey, cricket, football;
        public void init() {
                setLayout(new FlowLayout(FlowLayout.RIGHT));
                swim = new Checkbox("Swimming", null,  true);
                hockey = new Checkbox("Hockey");
                cricket = new Checkbox("Cricket");
                football = new Checkbox("Football");
                add(swim);
                add(hockey);
                add(cricket);
                add(football);
        }
}
```

Study the output of the above program as an exercise.

13.3.3 Border Layout :

The **BorderLayout** class implements a common layout style for top level windows. It has four narrow, fixed width components at the edges and one large area in the center. The four sides are referred to as north, south, east and west. The middle area is called the center. The constructors for **BorderLayout** are :

BorderLayout() BorderLayout(int *horz*, int *vert*)

The first form creates the default border layout. In the second form you can specify the horizontal and vertical space left between the components by *horz* and *vert* respectively.

BorderLayout defines the following constants which specify the regions :

BorderLayout.CENTER **BorderLayout.SOUTH**

BorderLayout.EAST **BorderLayout.WEST**

BorderLayout.NORTH

These constants are to be used while adding the components with the following form of **add()** which is defined by **Container** :

void add(Component *compObj*, Object *region*);

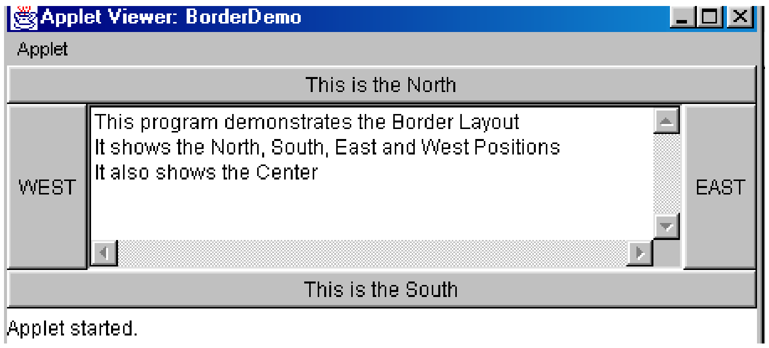
compObj is the component to be added and region specifies where the component will be added.

The following example demonstrates the **BorderLayout** :

```
import java.awt.*;
import java.applet.*;
import java.util.*;
/*
<applet code = “BorderDemo” width = 300 height = 200>
</applet>
*/
public class BorderDemo extends Applet {
    public void init() {
            setLayout(new BorderLayout());
            add(new Button(“This is the North”),
BorderLayout.NORTH);
            add(new Button(“This is the South”),
BorderLayout.SOUTH);
            add(new Button(“EAST”), BorderLayout.EAST);
            add(newButton(“WEST”), BorderLayout.WEST);
            String s = “This program demonstrates the
Border Layout \n” +

                    “It shows the North, South, East
    and West Positions \n” +
                    “It also shows the Center”;
            add(new TextArea(s),
BorderLayout.CENTER);
    }
}
```

The output of the above program is :



```
This program demonstrates the Border Layout
It shows the North, South, East and West Positions
It also shows the Center
```

Applet started.

13.3.1 to 13.3.3 Check Your Progress.

1. Write True or False.

a) Each **Container** object has a layout manager associated with it.

b) **BorderLayout** is the default Layout Manager.

13.3.4 Grid Layout :

This layout lays out the components in a two dimensional grid. You define the number of rows and columns when you instantiate the **GridLayout**. The constructors for **GridLayout** are :

GridLayout()   GridLayout(int *numRows*, int *numColumns*)

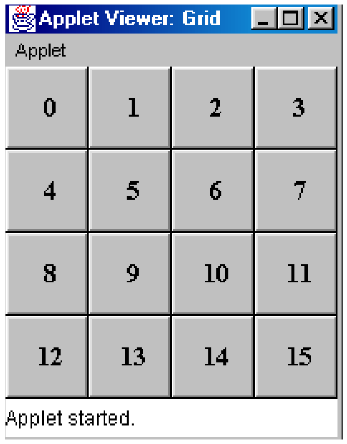
GridLayout(int *numRows*, int *numColumns*, int *horz*, int *vert*)

The first form creates a single column grid layout. With the second form you can create a grid layout of the specified number of rows and columns. The third form allows you to specify the horizontal and vertical space between components using the *horz* and *vert* respectively. Either *numRows* or *numCols* can be zero. If *numRows* is zero, it allows columns of unlimited length and specifying *numColumns* as zero allow for unlimited-length rows.

Study the following example that creates a grid layout of size 4 x 4 and fills it with buttons, where the button of each label is its index.

```
import java.awt.*;
import java.applet.*;
/*
<applet code = “Grid” width = 200 height = 200>
</applet>
*/
public class Grid extends Applet {
        static final int n = 4;
        public void init() {
                setLayout(new GridLayout(n, n));
                setFont(new Font(“Courier”, Font.BOLD, 16));
                for(int i = 0; i < n; i ++) {
                        for(int j = 0; j < n; j++) {
                                int k = i*n + j;
                                add(new Button(“ “ + k));
                        }
                }
        }
}
```

The output of the program will be :



13.3.5 Card Layout :

The **CardLayout** stores several different layouts. Each layout can be thought of as being on a separate index card in a deck that can be shuffled so that any card is on top at a given time.

CardLayout provides the following constructors :

CardLayout()   CardLayout(int *horz*, int *vert*)

In the first form, you create the default card layout. With the second form you can specify the horizontal and vertical spaces left between components by *horz* and *vert* respectively.

The cards are typically held in an object of type **Panel**. This panel must have a **CardLayout** as its layout manager. The cards which form the deck are also typically objects of type **Panel**. This means that you create a panel that contains the deck and a panel for each of the cards in the deck. Then, you must add the required components for each card. All the panels thus created must then be added to the panel for which the **CardLayout** is the layout manager. Finally, this panel is added to the main applet panel. After this is complete, you must allow the user some means to select between these cards. One common way is to provide a push button for each card in the deck.

You make use of **add()** to add cards to a panel.

The following form of **add()** is used :

void add(Component *panelObj*, Object *name*);

where *name* is the string that specifies the name of the card whose panel is specified by *panelObj*.

Once you have created a deck, a card is activated by your program by calling one of the following methods defined by **CardLayout** :

void first(Container *deck*)   void last(Container *deck*)

void next(Container *deck*)   void previous(Container *deck*)

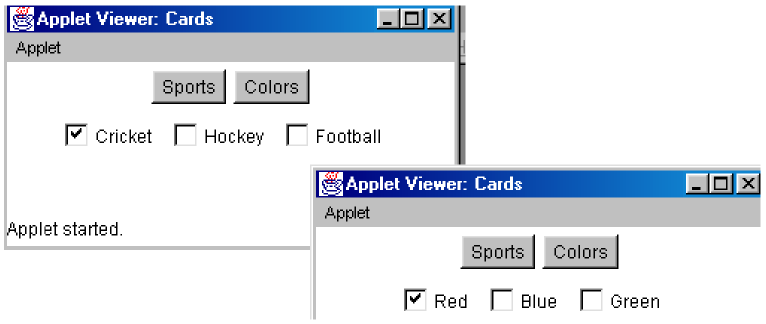
void show(Container *deck*, String *cardName*)

where *deck* is a reference to the container (usually a panel) which holds the cards, and *cardName* is the name of the card. **first()** causes the first card in the deck to be shown and **last()** causes the last card to be shown. To show the next card call **next()**. To show the previous card call **previous()**. The **show()** method displays the card whose name is passed in *cardName*.

The following example illustrates a two level card deck which allows the user to select either sports or colors each of which is displayed in a card. Each card is activated by pushing its button. By clicking the mouse you can cycle through the cards.

```
    import java.awt.*;
    import java.awt.event.*;
    import java.applet.*;
    /*
    <applet code = “Cards” width = 300 height = 100>
    </applet>
    */
    public void Cards extends Applet implements ActionListener, MouseListener
{
            Checkbox cricket, hockey, football, red, blue, green;
            Panel SportsColors;
            CardLayout clo;
            Button Sports, Colors;
            public void init() {
                    Sports = new Button(“Sports”);
                    Colors = new Button(“Colors”);
                    add(Sports);
                    add(Colors);
                    clo = new CardLayout();
                    SportsColors = new Panel();
                    SportsColors.setLayout(clo);
                    cricket = new Checkbox(“Cricket”, null, true);
                    hockey = new Checkbox(“Hockey”);
                    football = new Checkbox(“Football);
                    red = new Checkbox(“Red”, null, true);
                    blue = new Checkbox(“Blue”);
                    green = new Checkbox(“Green”);
                    Panel sPanel = new Panel();
                    sPanel.add(cricket);
                    sPanel.add(hockey);
```

```
                sPanel.add(football);
                Panel cPanel = new Panel();
                cPanel.add(red);
                cPanel.add(blue);
                cPanel.add(green);
                SportsColors.add(sPanel, “Sports”);
                SportsColors.add(cPanel, “Colors”);
                add(SportsColors);
                Sports.addActionListener(this);
                Colors.addActionListener(this);
                addMouseListener(this);
    }
    public void mousePressed(MouseEvent me) {
        clo.next(SportsColors);         // Cycle through cards
        }
                // empty implementations for other
    MouseListener methods
        public void mouseClicked(MouseEvent me) {
        }
        public void mouseEntered(MouseEvent me) {
        }
        public void mouseExited(MouseEvent me) {
        }
        public void mouseReleased(MouseEvent me) {
        }
        public void actionPerformed(ActionEvent ae) {
                if(ae.getSource() == Sports)
                        clo.show(SportsColors, “Sports”);
                else
                        clo.show(SportsColors, “Colors”):
        }
    }
```
output



13.3.4 to 13.3.5 Check Your Progress.

1. Answer the following.

a) Write down the constructors for a **GridLayout**.

____

____

b) Write the methods defined by **CardLayout** to activate a card.

____

____

13.4 SUMMARY

Controls are the components that allow the user to interact with your application. eg. a push button control. A layout manager automatically positions controls within a container. In addition to controls, a frame window can also include a menu bar. Each entry in a menu bar activates a drop down menu of options. A user can select the option of his choice. A menu bar is always positioned at the top of a window.

The controls supported by AWT are subclasses of Component and include the following : Labels, Push buttons, Check boxes, Check lists, Lists, Scroll bars, and Text editing.

Source : *books.google.co.in (Google book)*

13.5 CHECK YOUR PROGRESS - ANSWERS

13.1

1. a) False

b) False

c) True

13.2.1 & 13.2.2

1. a) setText

b) Button

c) getSource()

13.2.3 & 13.2.4

1. a) True

b) False

c) True

13.2.5 & 13.2.6

1. a) True

b) True

c) False

d) True

13.2.7 & 13.2.8

1. a) edit control

b) true

c) Scrollbars

d) 1

13.2.9

1. a) multiline

b) TextComponent

13.3.1 to 13.3.3

1. a) True

b) False

13.3.4 & 13.3.5

1. a) The constructors for **GridLayout** are: GridLayout(), GridLayout(int *numRows*, int *numColumns*),

GridLayout(int *numRows*, int *numColumns*, int *horz*, int *vert*)

b) The methods defined by **CardLayout** to activate a card are :

void first(Container *deck*), void last(Container *deck*), void next(Container *deck*), void previous(Container *deck*), void show(Container *deck*, String *cardName*)

13.6 QUESTIONS FOR SELF - STUDY

1. What do you understand by controls? List the controls supported by AWT. Explain how to include a control in a window.
2. Describe in detail any 5 controls supported by the AWT.
3. Describe the various methods of the **TextField** class.

4. **Write short notes on :**

   a) Scroll Bars

   b) TextArea

   c) GridLayout

   d) CardLayout

5. Describe the **FlowLayout** and **Border Layout** in detail.

13.7 SUGGESTED READINGS

1. www.java.com
2. www.freejavaguide.com
3. www.java-made-easy.com
4. www.tutorialspoint.com
5. www.roseindia.net

❑ ❑ ❑

Notes

Notes

Chapter : 14

Introduction to Swing

14.0 OBJECTIVES

Dear students,

After studying this chapter you will able to :

- discuss Swing components like buttons, checkboxes and labels
- discuss components like tabbed panes, trees and tables

14.1 INTRODUCTION

Swing is a set of classes which provide more flexible and powerful components than those that are possible in the AWT. Swing provides the components such as buttons, check boxes and labels, but additionally it also provides components like tabbed panes, scroll panes, trees and tables. Here we present only a brief overview of **Swing**.

We know that the AWT components are implemented by platform specific code. Swing components on the other hand, are written entirely in Java and therefore are platform independant. The term *lightweight* is used to describe such elements. Fundamental to **Swing** is the **JApplet** class which extends **Applet**. **JApplet** is rich with functionality that is not provided by **Applet**. Remember however, that when adding a component to an instance of **JApplet**, do not invoke the **add()** method of the applet. Instead call **add()** of the content pane of the **JApplet** object. The content pane is obtained by the following method :

Container getContentPane()

To add a component we use the method **add()** of **Container** as :

void add(*comp*)

where *comp* is the component to be added.

14.2 ICONS AND LABLES

14.2.1 Icons

In **Swing**, icons are encapsulated by the **ImageIcon** class, which paints an icon from an image. Its constructors are :

ImageIcon(String *filename*) ImageIcon(URL *url*)

The first form of the constructor uses the image in the file named *filename*. The second form uses the image in the resource identified by *url*. **ImageIcon** class implements the **Icon** interface which declares the following methods :

Method	**Description**
int getIconHeight()	returns the height of the icon in pixels
int getIconWidth()	returns the width of the icon in pixels
void paintIcon(Component *comp*,Graphics *g,* int *x*, int *y*)	paints the icon at position *x*, *y* on the graphics context *g*.

14.2.2 Labels

Swing labels are instances of the **JLabel** class which extends **JComponent**. It can display text and/or icon. Some of its constructors are :

JLabel(Icon *i*) JLabel(String *s*)

JLabel(String *s*, Icon *i*, int *align*)

where *s* and *i* are the text and icon used for the label. The **align** argument is either **LEFT**, **RIGHT** or **CENTER**. These constants are defined in the **SwingConstants** interface.

The methods with which the icons and the text associated with the label can be read and written are :

Icon getIcon() String getText()

void setIcon(Icon *i*) void setText(String *s*)

where *i* and *s* are icon and text respectively.

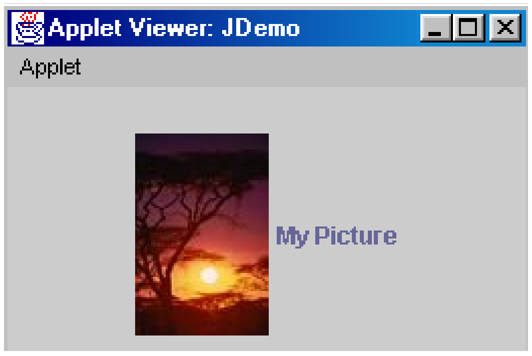
Let us see how to display text and icon with the following example :

```
import java.awt.*;
import javax.swing.*;
```

```
/*
<applet code = “JDemo” width = 300 height = 60>
</applet>
*/
public class JDemo extends JApplet {
        public void init() {
        Container cp = getContentPane();    // get content
                  pane
        ImageIcon ii = new ImageIcon(“pic1.gif”);
                     // create an icon
        JLabel jl = new JLabel(“My Picture, ii, JLabel.CENTER);   // create
label

             cp.add(jl);        // Add label to content pane
        }
}
```

output



14.1 & 14.2 Check Your Progress.

1. Write True or False.

a) Swing components are lightweight.

b) Icons are instances of the JLabel class.

c) Applet class extends JApplet.

14.3 TEXT FIELD

JTextComponent encapsulates the Swing text field and extends **JComponent**. One of its subclasses is the **JTextField** which allows you to edit a single line of text. Some of its constructors are :

JTextField()   JTextField(int *cols*)

JTextField(String *s*, int *cols*)   JTextField(String *s*)

s is the string to be presented and *cols* is the number of columns in the text field.

The following example illustrates :

```
            import java.awt.*;
            import javax.swing.*;
            /*
            <applet code = “JTextFieldDemo” width = 300 height = 50>
            </applet>
            */
            public class JTextFieldDemo extends JApplet {
   JTextField jtf;
   public void init() {
          Container cp = getContentPane();
          cp.setLayout(new FlowLayout());          // set flow layout
          jtf = new JTextField(15);        // set text field
          cp.add(jtf);            // add to content pane
   }
}
```

Run the program and study how to obtain the text field.

14.4 BUTTONS

Swing buttons are subclasses of the **AbstractButton** class, which extends **JComponent**. This class contains many methods which allow you to control the behaviour of buttons, checkboxes, and radio buttons. It is possible to associate an icon with a swing button. You can define different icons that are displayed for a component when it is disabled, pressed or selected. Another icon can be used as a rollover icon which is displayed when the mouse is positioned over that component. The methods which control this behaviour are:

void setDisabledIcon(Icon *di*)   void setPressedIcon(Icon *pi*)

void setSelectedIcon(Icon *si*)   void setRolloverIcon(Icon *ri*)

where *di*, *pi*, *si* and *ri* are the icons to be used for these different conditions. The text associated with the button can be read and written with the following methods :

String getText()

void setText(String *s*)

respectively, where *s* is the text to be associated with the button.

Concrete subclasses of **AbstractButton** generate action events when they are pressed. The methods to register and unregister for these events are :

void addActionListener(ActionListener *al*)

void removeActionListener(ActionListener *al*)

where *al* is the action listener.

AbstractButton is the superclass for push buttons, check boxes and radio buttons. Let us see each of these in the following sections :

The JButton Class : This class allows an icon, a string or both to be associated with a push button. Some of its constructors are :

JButton(Icon *i*)   JButton(String *s*)

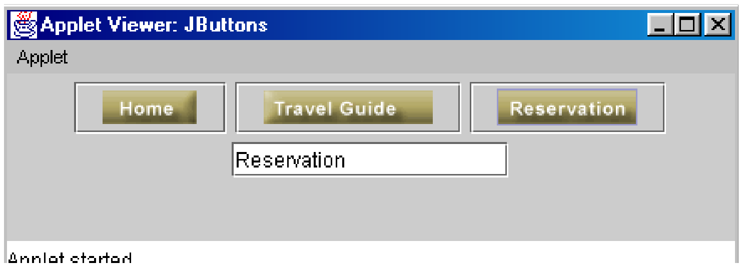
JButton(String *s*, Icon *i*)

where *s* and *i* are string and icon used for the button respectively.

The following program demonstrates use of buttons. It displays three pushbuttons and a text field. Each pushbutton displays a menu item. The applet is created by getting the content pane and setting the layout manager for the pane. Image buttons for menu items are created and added to the pane. The applet is then registered to receive action events that are generated by the buttons. A text field is also created and added to the applet. The handler for action events displays the command string, which is associated with the button in the text field.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code = “JButtons” width = 300 height = 50>
</applet>
*/
public class JButtons extends JApplet implements ActionListener {
                JTextField jtf;
                public void init() {
                        Container cp = getContentPane();
                        cp.setLayout(new FlowLayout());
                        ImageIcon home = new
                        ImageIcon(“Home.gif”);
                        JButton jb = new JButton(home);
                        jb.setActionCommand(“Home”);
                        jb.addActionListener(this);
                        cp.add(jb);
                        ImageIcon travel = new
                        ImageIcon(“Travel.gif”);
                        JButton jb1 = new JButton(travel);
                jb1.setActionCommand("Travel");
                jb1.addActionListener(this);
                cp.add(jb1);
                ImageIcon reserve = new
        ImageIcon("Reservation.gif");
                JButton jb2 = new JButton(reserve);
                jb2.setActionCommand("Reservation");
                jb2.addActionListener(this);
                cp.add(jb2);
                jtf = new JTextField(15);
                cp.add(jtf);
        }
        public void actionPerformed(ActionEvent ae) {
                jtf.setText(ae.getActionCommand());
        }
}
```

output



14.3 & 14.4 Check Your Progress.

1. Fill in the blanks.

a) .... allows you to edit a single line of text in Swing.

b) .... is the superclass for push buttons, check boxes and radio buttons.

c) A .... icon is displayed when the mouse is positioned over a component.

14.5 CHECKBOXES

The **JCheckBox** class provides the functionality of the check box and it is a concrete implementation of **AbstractButton**. Some of its constructors are :

JCheckBox(Icon *i*)	JCheckBox(Icon i, boolean *state*)
JCheckBox(String *s*)	JCheckBox(String *s*, boolean *state*)

JCheckBox(String *s*, Icon *i*)   JCheckBox(String *s*, Icon *i*, boolean *state*)

where *i* is the icon for the button, *s* specifies the text. If state is **true**, then the checkbox is initially selected, otherwise it is not. The state of the checkbox can be changed with the following method :

void setSelected(boolean *state*)

state is **true** if the check box should be checked.

Let us create an applet to display check boxes and a text field. When a checkbox is pressed, its text will be displayed in the text field. Obtain the content pane for the **JApplet** object and set the layout. Then add check boxes to the content pane. Register the applet to receive the item events. Finally add a text field to the content pane. Whenever a check box is slected or deselected an item event is generated. This is handled by the **itemStateChanged()** method. Inside **itemStateChanged()**, the **getItem()** method will get the **JCheckBox** object which generated the event. Then the **getText()** method gets the text for that check box and sets the text inside the text field.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
    <applet code = “JCheckBoxes” width = 350 height = 40>
                    </applet>
*/
public class JCheckBoxes extends JApplet implements ItemListener {
                    JTextField jtf;
                    public void init() {
                        Container cp = getContentPane();
                        cp.setLayout(new FlowLayout());
                        JCheckBox cb = new
                        JCheckBox(“Swimming”);
                        cb.addItemListener(this);
                        cp.add(cb);
                        JCheckBox cb1 = new
                        JCheckBox(“Skating”, true);
                        cb1.addItemListener(this);
                        cp.add(cb1);
                        JCheckBox cb2 = new
                        JCheckBox(“Riding”);
                        cb2.addItemListener(this);
                        cp.add(cb2);
```

```
                jtf = new JTextField(15);
                cp.add(jtf);
        }
        public void itemStateChanged(ItemEvent ie) {
                JCheckBox cb = (JCheckBox)ie.getItem();
                jtf.setText(cb.getText());
        }
}
```

Study the output of the program. Also modify the program to set icons for a component when it is selected, disabled, rollover or pressed.

14.6 RADIO BUTTONS

Radio Buttons are supported by the **JRadioButton** class, which is a concrete implementation of **AbstractButton**. Some of its constructors are :

JRadioButton(Icon *i*)

JRadioButton(Icon *i*, boolean *state*)

JRadioButton(String *s*)

JRadioButton(String *s*, boolean *state*)

JRadioButton(String *s*, Icon *i*)

JRadioButton(String *s*, Icon *i*, boolean *state*)

where *i* is the icon for the button, *s* specifies the text. If *state* is **true**, the button is initially selected, otherwise not.

Radio buttons should be configured in a group. As you know, only one button in that group can be selected at any time. If a user presses a radio button in a group, then any previously selected button gets deselected. In order to create a button group, we use the class **ButtonGroup**. Its default constructor is invoked to create the group and buttons are then added to the group with the method

void add(AbstractButton *ab*)

where *ab* is the reference to the button to be added to the group.

Radio button presses generate action events. In the following example, these events are handled by the **actionPerformed()** method. The **getActionCommand()** method gets the text associated with the button and uses it to set the text field.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code =”JRadio” width = 300 height = 50>
</applet>
*/
```

```
public class JRadio extends JApplet implements ActionListener {
        JTextField jtf;
        public void init() {
                Container cp = getContentPane();
                cp.setLayout(new FlowLayout());
                JRadioButton b1 = new JRadioButton("A");
                b1.addActionListener(this);
                cp.add(b1);
                JRadioButton b2 = new JRadioButton("B");
                b2.addActionListener(this);
                cp.add(b2);
                JRadioButton b3 = new JRadioButton("C");
                b3.addActionListener(this);
                cp.add(b3);
                ButtonGroup bg = new ButtonGroup();
                bg.add(b1);
                bg.add(b2);          // create a button group
                bg.add(b3);
                jtf = new JTextField(15);
                cp.add(jtf);
        }
        public void actionPerformed(ActionEvent ae) {
                jtf.setText(ae.getActionCommand());
        }
}
```

Study the program and its output.

14.7 COMBE BOXES

Swing provides a combination of a text field and a drop down list called a combo box through the **JComboBox** class, which extends **JComponent**. Normally a combo box displays one entry. But it can also display a drop down list which allows the user to select a different entry. Your own selection can also be typed in the text field.

Some constructors of **JComboBox()** are :

JComboBox()   JComboBox(Vector *v*)

where *v* is a vector which initialises the combo box.

Items are added to the list using the **addItem()** method whose form is :

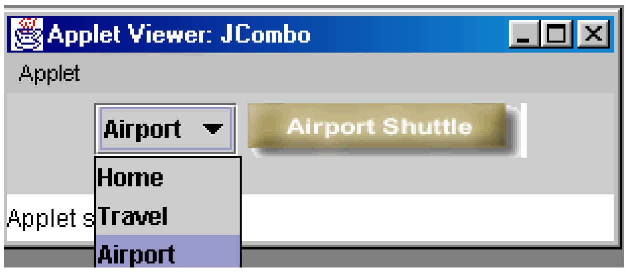
void addItem(Object *obj*)

where *obj* is the item to be added to the combo box.

Let us create a combo box and a label in the following example. The label displays an icon. The combo box contains entries from a menu list. Whenever an item is selected the label is updated to display the image of that item.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code = “JCombo” width = 300 height = 50>
</applet>
*/
public class JCombo extends JApplet implements ItemListener {
    JLabel jl;
    ImageIcon home, travel, airport;
    public void init() {
            Container cp = getContentPane();
            cp.setLayout(new FlowLayout());
            JComboBox jc = new JComboBox();
            jc.addItem(“Home”);
            jc.addItem(“Travel”);
            jc.addItem(“Airport”);
            jc.addItemListener(this);
            cp.add(jc);
            jl = new JLabel(new ImageIcon(“home.gif”);
            cp.add(jl);
    }
    public void itemStateChanged(ItemEvent ie) {
    String s = (String)ie.getItem();
    jl.setIcon(new ImageIcon(s + “.gif”));
    }
}
```

The output of the program is as shown below :



14.5 to 14.7 Check Your Progress.

1. Match the following.

Column A	**Column B**
a) Combo Boxes	i) a combination of a text field and a drop down list
b) Radio Buttons	ii) provided in the JCheckBox class
c) Check Boxes	iii) should be configured in a group.

14.8 TABBED PANES

A tabbed pane is a component which appears like a group of folders in a file cabinet. Each folder has a title. When a user selects a folder, its contents become visible. At any time, only one of the folders may be selected. Tabbed panes are mainly used to set configuration options. The **JTabbedPane** class encapsulates tabbed panes. It extends the **JComponent**.

void addTab(String *str*, Component *comp*)

is the method used to define tabs. Here *str* is the title for the tab and *comp* is the component that should be added to the tab. Typically a **JPanel** or its subclass is added.

Following is the procedure to use a tabbed pane in an applet :

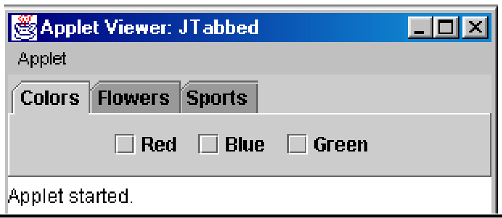
- Create a **JTabbedPane** object
- Call **addTab()** to add a tab to the pane.
- Repeat the above step for each tab
- Add the tabbed pane to the content pane of the applet.

In the following example we shall create three tabs in a tabbed pane. The first tab is titled "Colors" and contains three check boxes.The second tab contains buttons to show names of flowers and the tab is titled"Flowers". The third tab titled "Sports" contains a combo box, which allows to select a sport from one of the three sports.

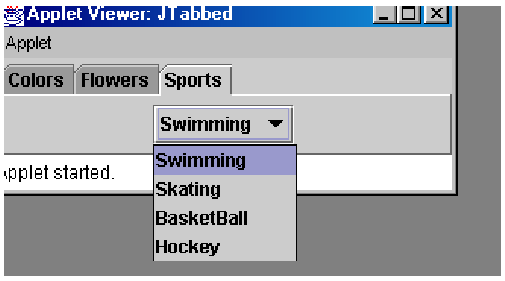
```
import javax.swing.*;
/*
<applet code = "JTabbed" width = 300 height = 50>
</applet>
*/
public class JTabbed extends JApplet {
                    public void init() {
                        JTabbedPane jp = new JTabbedPane();
                        jp.addTab("Colors", new ColorPanel());
                        jp.addTab("Flowers", new FlowersPanel());
                        jp.addTab("Sports", new SportsPanel());
                        getContentPane.add(jp);
```

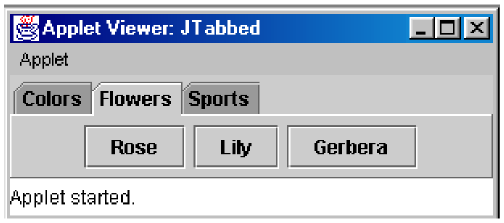
```
        }
}
class ColorPanel extends JPanel {
        public ColorPanel() {
                    JCheckbox cb1 = new JCheckBox(“Red”);
                    add(cb1);
                    JCheckBox cb2 = new JCheckBox(“Blue”);
                    add(cb2);
                    JCheckBox cb3 = new JCheckBox(“Green”);
                    add(cb3);
        }
}
class FlowersPanel extends JPanel {
        public FlowersPanel() {
                    JButton b1 = new JButton(“Rose”);
                    add(b1);
                    JButton b2 = new JButton(“Lily”);
                    add(b2);
                    JButton b3 = new JButton(“Gerbera”);
                    add b3;
        }
}
class SportsPanel extends JPanel {
        public SportsPanel() {
                    JComboBox jc = new JComboBox();
                    jc.addItem(“Swimming”);
                    jc.addItem(“Skating”);
                    jc.addItem(“BasketBall”);
                    jc.addItem(“Hockey”);
        }
}
```

The output of the program is as follow :



Applet started.





14.9 SCROLL PANES

A scroll pane is a component that presents a rectangular area in which a component may be viewed. If necessary, horizontal and/or vertical scroll bars may be provided. Scroll panes are implemented using the **JScrollpane** class, which extends **JComponent**. Some of its constructors are :

JScrollPane(Component *comp*)

JScrollPane(int *vsb*, int *hsb*)

JScrollPane(Component *comp*, int *vsb*, int *hsb*)

where *comp* is the component ot be added to the scroll pane, *vsb* and *hsb* are **int** constants that define when vertical and horizontal scroll bars for this scroll pane are shown. These constants are defined by the **ScrollPaneConstants** interface.

Some of these constants are :

Constant	**Description**
HORIZONTAL_SCROLLBAR_ALWAYS	always provide horizontal scroll bar
VERTICAL_SCROLLBAR_ALWAYS	always provide vertical scroll bar
HORIZONTAL_SCROLLBAR_AS_NEEDED	provide horizontal scroll bar if needed
VERTICAL_SCROLLBAR_AS_NEEDED	provide vertical scroll bar if needed

The steps to be followed to use a scroll pane in an applet :

- Create a **JComponent** object
- Create a **JScrollPane** object
- Add the scroll pane to the content pane of the applet.

The following example illustrates. We add a **JPanel** object to the scroll pane and the scroll pane is added to the content pane. We add buttons to the **Jpanel** object. Vertical and horizontal scroll bars will appear if the buttons do not fit in the dimensions. These can be used to scroll the buttons into view.

```
import java.awt.*;
import javax.swing.*;
/*
<applet code = “JScrolls” width = 300 height = 50>
</applet>
*/
public class JScrolls extends JApplet {
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new BorderLayout());
        JPanel jp = new JPanel();
        jp.setLayout(new GridLayout(20,20));
        int b = 0;
        for(int i = 0; i < 5; i++) {
            for(int j = 0; j<5; j++) {
            jp.add(new JButton(“Button “ + b));
                b++;
            }
        }
int v = ScrollPaneConstants.
VERTICAL_SCROLLBAR_AS_NEEDED;
int h = SrollPaneConstants.
HORIZONTAL_SCROLLBAR_AS_NEEDED;
        JScrollPane jsp = new JScrollPane(jp, v, h);
        cp.add(jsp, BorderLayout.CENTER);
    }
}
```

Run the program and study its output. Also try adding more buttons to see effect of the scrollbars.

14.10 TREES

A tree is a component which presents a hierarchical view of data. A user has the ability to expand or collapse indivisual subtrees in this display. Trees are implemented in **Swing** by the **JTree** class which extends **JComponent**. Some of its constructors are :

JTree(Hashtable *ht*)

This form creates a tree in which each element of the hashtable *ht* is a child node.

JTree(Object *obj*[])

In this form each element of the array *obj* is a child node.

JTree(TreeNode *tn*)

The tree node *tn* is the root of the tree in this form.

JTree(Vector *v*)

Elements of vector *v* are used as child nodes.

A **Jtree** object generates events whenever a node is exapanded or collapsed. The **addTreeExpansionListener()** and **removeTreeExpansionListener()** methods allow listeners to register and unregister for these notifications.

The signatures of these methods are :

void addTreeExpansionListener(TreeExpansionListener *tel*)

void removeTreeExpansionListener(TreeExpansionListener *tel*)

where *tel* is the listener object.

The **getPathForLocation()** method is used to translate a mouse click on a specific point of the tree to a tree path. Its signature is :

TreePath getPathForLocation(int *x*, int *y*)

where *x* and *y* are the coordinates at which the mouse is clicked. The return value is a **TreePath** object that encapsulates information about the tree node that was selected by the user. This class provides several constructors and methods. We shall use only the **toString()** method which returns the string equivalent of the tree path.

The **TreeNode** interface declares methods that obtain information about a tree node eg. it is possible to obtain a reference to the parent node or the number of child nodes. The **MutableTreeNode** is an interface that extends **TreeNode**. It declares methods which can insert and remove child nodes or change the parent node. The **DefaultMutableTreeNode** class implements the **MutableTreeNode** interface. It represents a node in a tree.

One of its constructors is :

DefaultMutableTreeNode(Object *obj*)

where *obj* is the object enclosed in this tree node. The new tree node does not have a parent or children.

In order to create a hierarchy of tree nodes, there is a method **add()** in **DefaultMutableTreeNode**.

Its signature is :

void add(MutableTreeNode *child*)

child is a mutable tree node that is to be added as a child to the current node.

The **TreeExpansionEvent** class in the **javax.swing.event** package describes the tree expansion events. **getPath()** method of this class returns a

TreePath object which describes the path to the changed node.

The **TreeExpansionListener** interface provides the following methods :

void treeCollapsed(TreeExpansionEvent *tee*)

void treeExpanded(TreeExpansionEvent *tee*)

tee is the tree expansion event. The first method is called when a subtree is hidden and the second method is called when a subtree becomes visible.

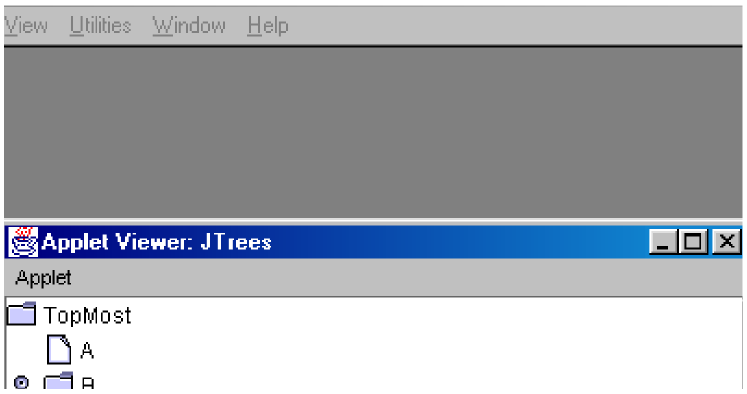
The steps to use a tree in an applet are :

- Create a **JTree** object
- Create a **JScrollPane** object
- Add the tree to the scroll pane
- Add the scroll pane to the content pane of the applet.

The following example illustrates. In this example a **DefaultMutableTreeNode** labeled “TopMost” is created. This is the top node in the hierarchy. Additional tree nodes are created and connected to this node with the **add()** method. A reference to the top node is provided as an argument to the **JTree** constructor. The **JTree** is then provided as argument to **JScrollPane** constructor which is then added to the applet. A textfield is also created and added to the applet to present information about mouse click. For the purpose of receiving mouse events, the **addMouseListener()** method of the **JTree** object is called. The **doMouseClicked()** method processes mouse clicks. It calls the **getPathForLocation()** which translates the coordinates of the mouse click into a **TreePath** object.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.tree.*;
/*
<applet code = “JTrees” width = 400 height = 200>
</applet>
*/
public class JTrees extends JApplet {
        JTree tree;
        JTextField jtf;
        public void init() {
      Container cp = getContentPane();
      cp.setLayout(new BorderLayout());
//Create top node of tree
        DefaultMutableTreeNode top = new
```

```
DefaultMutableTreeNode(“Topmost”);
//Create subtree A
                            DefaultMutableTreeNode a = new
                            DefaultMutableTreeNode(“A”);
                                  top.add(a);
                            DefaultMutableTreeNode a1 = new
                            DefaultMutableTreeNode(“A1”);
                            a.add(a1);
                            DefaultMutableTreeNode a2 = new
                            DefaultMutableTreeNode(“A2”);
                            a.add(a2);
                            DefaultMutableTreeNode a3 = new
    DefaultMutableTreeNode(“A3”);
                            a.add(a3);
                            //Create subtree B
                            DefaultMutableTreeNode b = new
                            DefaultMutableTreeNode(“B”);
                            top.add(b);
                            DefaultMutableTreeNode b1 = new
                            DefaultMutableTreeNode(“B1”);
                            b.add(a1);
                            DefaultMutableTreeNode b2 = new
                            DefaultMutableTreeNode(“B2”);
                            b.add(a2);
                            DefaultMutableTreeNode b3 = new
                            DefaultMutableTreeNode(“B3”);
                            b.add(a3);
                            tree = new JTree(top);
// Create tree
    int v = ScrollPaneConstants.
VERTICAL_SCROLLBAR_AS_NEEDED;
    int h = rollPaneConstants.
HORIZONTAL_SCROLLBAR_AS_NEEDED;
                            JScrollPane jsp = new JScrollPane(tree, v, h);
                                  cp.add(jsp, BorderLayout.CENTER);
                                  jtf = new JTextField(“ “, 20);
                                  cp.add(jtf, BorderLayout.SOUTH);
                                  tree.addMouseListener(new
                        MouseAdapter() {
                       public void mouseClicked(MouseEvent me) {
                                   doMouseClicked(me) ;
                             }
                        } );
                 }
                        void doMouseClicked(MouseEvent me) {
                        TreePath tp =
                 tree.getPathForLocation(me.getX(),
             me.getY());
                              if(tp != null)
                                   jtf.setText(tp.toString());
                              else
                                   jtf.setText(" ");
                        }
                 }
             output
```



14.11 TABLES

A table is a component which displays rows and columns of data.You can resize the columns by dragging the cursor on the column boundaries. You can also drag a column to a new position. Tables are implemented in the **JTable** class which extends **JComponent**. One of its constructors is :

JTable(Object *data*[][], Object *colHeads*[])

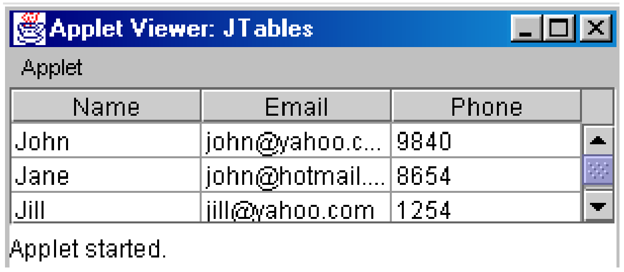
where *data* is a two dimensional array of the information to be presented and *colHeads* is a one dimensional array with the column headings. The steps for using a table in an applet are :

- Create a **JTable** object.
- Create a **JScrollPane** object.
- Add the table to the scroll pane.
- Add the scroll pane to the content pane of the applet.

Let us see how the following example implements a table in the applet. A one dimensional array of headings is created and a two dimensional arry of data is created. The arrays are passed to the **JTable** constructor. The table is added to the scroll pane and the scroll pane is added to the content pane.

```
import java.awt.*;
import javax.swing.*;
/*
<applet code = “JTables” width = 300 height = 100>
</applet>
*/
public class JTables extends JApplet {
    public void init() {
            Container cp = getContentPane();
            cp.setLayout(new BorderLayout());
            final String[] colHeads = {“Name”, “Email”,
“Phone”);
            final Object[][] data = {
               { “John”, “john@yahoo.com”, “9840”},
                { “Jane”, “jane@hotmail.com”, “8654”},
                 { “Jill”, “jill@yahoo.com”, “1254”},
                { “Jack”, “jack@yahoo.com”, “6534”},
                 { “Jerry”, “jerry@hotmail.com”, “7433”},
            { “Joseph”, “joseph@hotmail.com”, “4343”},
            };
    JTable table = new JTable(data, colHeads);
                    // create table
    int v = ScrollPaneConstants.
VERTICAL_SCROLLBAR_AS_NEEDED;
    int h = rollPaneConstants.
HORIZONTAL_SCROLLBAR_AS_NEEDED;
JScrollPane jsp = new JScrollPane(table, v, h);
                            // add table to scroll pane
    cp.add(jsp, BorderLayout.CENTER);
                    // add scroll pane to contentpane
    }
}
```

The output of the program is :



Name	Email	Phone
John	john@yahoo.c...	9840
Jane	john@hotmail....	8654
Jill	jill@yahoo.com	1254

Applet started.

14.8 to 14.11 Check Your Progress.

1. Match the following.

Column A	**Column B**
a) Trees	i) a component that presents a rectangular area
b) Tables	ii) component which appears like group of folders
c) Tabbed pane	iii) displays rows and columns of data
d) Scroll pane	iv) component which presents hierarchical view of data

14.12 SUMMARY

Swing is a set of classes which provides more flexible and powerful components than those that are possible in the AWT. Fundamental to Swing is the JApplet class which extends Applet.

A tabbed pane is a component which appears like a group of folders in a file cabinet. A tree is a component which presents a hierarchical view of data. A user has the ability to expand or collapse indivisual subtrees in this display. The TreeNode interface declares methods that obtain information about a tree node eg. it is possible to obtain a reference to the parent node or the number of child nodes.

Source : *sasajava.blogspot.com (Link)*

14.13 CHECK YOUR PROGRESS - ANSWERS

14.1 & 14.2

1. a) True

b) False

c) False

14.3 & 14.4

1. a) JTextField

b) AbstractButton

c) rollover

14.5 to 14.7

1. a) - (i)

b) - (iii)

c) - (ii)

14.8 to 14.11

1. a) - (iv)

b) - (iii)

c) - (ii)

d) - (i)

14.14 QUESTIONS FOR SELF - STUDY

1. Describe some methods of the **AbstractButton** class. Write a few constructors of the **JButton** class.
2. Describe the constructors of the **JCheckBox** class and the **JRadioButton** class. Which is the method used to add buttons to a group?
3. What is the meaning of a combo box? Write constructors of the combo box.
4. What is a tabbed pane? Write the procedure to add a tabbed pane to an applet.
5. What is a scroll pane? Describe the constructors of the **JScrollPane** class. Write down the steps to use **scroll** panes in an applet.
6. Write a detailed note on tree component of **Swing**.
7. What is the table component? Describe its constructors. Write the steps for using a table in an applet.
8. **Write short notes on :**

   a) Icons in Swing

   b) Labels in Swing

   c) TextFields in Swing

14.15 SUGGESTED READINGS

1. www.java.com
2. www.freejavaguide.com
3. www.java-made-easy.com
4. www.tutorialspoint.com
5. www.roseindia.net

❑ ❑ ❑

Notes