MADHYA PRADESH BHOJ(OPEN) UNIVERSITY BHOPAL

Java Programming

BSDS-301

Unit -1
Introduction to Java

Structure of the Unit

1.0 Objective

After going through this unit, you will have overview of the followings:

- Introduction to Object Oriented Programming (OOP)
- Characteristics of OOP
- Difference between object oriented and procedural approaches
- applications of OOP
- Understand the benefits of OO approach.

1.1 Introduction

The earliest computers were programmed in machine language using 0 and 1. The mechanical switches were used to load programs. Then, to provide convenience to the programmer, assembly language was introduced where programmers use pneumonic for various instructions to write programs. But it was a tedious job to remember so many pneumonic codes for various instructions. Other major problem with the assembly languages is that they are machine architecture dependent.

To overcome the difficulties of Assembly language, high-level languages came into existence. Programmers could write a series of English-like instructions that a compiler or interpreter could translate into the binary language of computers directly.

These languages are simple in design and easy to use because programs at that time were relatively simple tasks like any arithmetic calculations. As a result, programs were pretty short, limited to about a few hundred line of source code. As the capacity and capability of computers increased, so did the scope to develop more complex computer programs. However, these languages suffered the limitations of reusability, flow control (only goto statements), difficulty due to global variables, understanding and maintainability of long programs.

Structured Programming

When the program becomes larger, a single list of instructions becomes unwieldy. It is difficult for a programmer to comprehend a large program unless it is broken down into smaller units. For this reason languages used the concept of functions (or subroutines, procedures, subprogram) to make programs more comprehensible.

A program is divided into functions or subroutines where each function has a clearly defined purpose and a defined interface to the other functions in the program. Further, a number of functions are grouped together into larger entity called a module, but the principle remains the same, i.e. a grouping of components that carry out specific tasks.

Dividing a program into functions and modules is one of the major characteristics of structured programming. By dividing the whole program using functions, a structured program minimizes the chance that one function will affect another. Structured programming helps the programmer to write an error free code and maintain control over each function.

This makes the development and maintenance of the code faster and efficient. Structured programming remained the leading approach for almost two decades. With the emergence of new applications of computers the demand for software arose with many new features such as GUI (Graphical User Interface). The complexity of such programs increased multi-fold and this approach started showing new problems.

The problems arose due to the fundamental principle of this paradigm. The whole emphasis is on doing things. Functions do some activity, maybe a complex one, but the emphasis is still on doing. Data are given a lower status. For example in banking application, more emphasis is given to the function which collects the correct data in a desired format or the function which processes it by doing some summation, manipu-lation etc. or a function which displays it in the desired format or creates a report. But you will also agree that the important part is the data itself.

The major drawback with structured programming are its primary components, i.e., functions and data structures. But unfortunately functions and data structures do not model the real world very well. Basically to model a real world situation data should be given more importance. Therefore, a new approach emerges with which we can express solutions in terms of real world entities and give due importance to data.

The world is Object Oriented, and Object Oriented programming expresses programs in the ways that model how people perceive the world, Different real world objects around us which we often use for performing different functions. This shows that problem solving using the objects oriented approach is very close to our real life problem solving techniques.

The basic difference in Object Oriented programming (OOP) is that the program is organized around the data being operated upon rather than the operations performed. The basic idea behind OOP is to combine both, data and its functions that operate on the data into a single unit called object.

1.2 Introduction to Object Oriented Programming

Object Oriented Programming or OOP is the technique to create programs based on the real world. Unlike procedural programming, here in the OOP programming model programs are organized around objects and data rather than actions and logic. Objects represent some concepts or things and like any other objects in the real Objects in programming language have certain behavior, properties, type, and identity. In OOP based language the principal aim is to find out the objects to manipulate and their relation between each other. OOP offers greater flexibility and compatibility and is popular in developing larger application. Another important work in OOP is to classify objects into different types according to their properties and behavior. So OOP based software application development includes the analysis of the problem, preparing a solution, coding and finally its maintenance.

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

Class - It is the central point of OOP and that contains data and codes with behavior. In Java everything happens within class and it describes a set of objects with common behavior. The class definition describes all the properties, behavior, and identity of objects present within that class. As far as types of classes are concerned, there are predefined classes in languages like C++ and Pascal. But in Java one can define his/ her own types with data and code.

Object - Objects are the basic unit of object orientation with behavior, identity. As we mentioned above, these are part of a class but are not the same. An object is expressed by the variable and methods within the objects. Again these variables and methods are distinguished from each other as instant variables, instant methods and class variable and class methods.

Methods - We know that a class can define both attributes and behaviors. Again attributes are defined by variables and behaviors are represented by methods. In other words, methods define the abilities of an object.

Inheritance - This is the mechanism of organizing and structuring software program. Though objects are distinguished from each other by some additional features but there are objects that share certain things common. In object oriented programming classes can inherit some common behavior and state from others. Inheritance in OOP allows to define a general class and later to organize some other classes simply adding some details with the old class definition. This saves work as the special class inherits all the properties of the old general class and as a programmer you only require the new features. This helps in a better data analysis, accurate coding and reduces development time.

Abstraction - The process of abstraction in Java is used to hide certain details and only show the essential features of the object. In other words, it deals with the outside view of an object (interface).

Encapsulation - This is an important programming concept that assists in separating an object's state from its behavior. This helps in hiding an object's data describing its state from any further modification by external component. In Java there are four different terms used for hiding data constructs and these are public, private, protected and package. As we know an object can associated with data with predefined classes and in any application an object can know about the data it needs to know about. So any unnecessary data are not required by an object can be hidden by this process. It can also be termed as information hiding that prohibits outsiders in seeing the inside of an object in which abstraction is implemented.

Polymorphism - It describes the ability of the object in belonging to different types with specific behavior of each type. So by using this, one object can be treated like another and in this way it can create and define multiple level of interface. Here the programmers need not have to know the exact type of object in advance and this is being implemented at runtime.

1.2.1 Objects and Classes

Classes are the single most important feature of Java. Everything in Java is either a class, a part of a class, or describes how a class behaves. Although classes will be covered in great detail in Section Four, they are so fundamental to an understanding of Java programs that a brief introduction is going to be given here.

All the action in Java programs takes place inside class blocks, in this case the HelloWorld class. In Java almost everything of interest is either a class itself or belongs to a class. Methods are defined inside the classes they belong to. This may be a little confusing to C++ programmers who are used to defining all but the simplest methods outside the class block, but this approach is really more sensible. C++ takes the road it does primarily out of a desire to be compatible with C, not out of good object-oriented design. Both syntactically and logically everything in Java happens inside a class.

Even basic data primitives like integers often need to be incorporated into classes before you can do many useful things with them. The class is the fundamental unit of Java programs, not source code files like in C. For instance consider the following Java program:

```
classHelloWorld {
public static void main (String args[ ]) {
System.out.printIn(Hello World);
}
}
classGoodbyeWorld {
public static void main (String args[ ]) {
System.out.printIn(Goodbye World!);
}
}
```

Save this code in a single file called hellogoodbye.java in your java html directory, and compile it with the command javac hellogoodbye.java. Then list the contents of the directory. You will see that the compiler has produced two separate class files, HelloWorld.class and Goodbye World.class.

The second class is a completely independent program. Type java GoodbyeWorld and then type java HelloWorld. These programs run and execute independently of each other although they exist in the same source code file. Off the top of my head I cannot think of why you might want two separate programs in the same file, but if you do the capability is there.

It is more likely that you will want more than one class in the same file. In fact, you will see source code files with many classes and methods.

1.3 Characteristic of OOP

OOP offers several benefits to both the program developer and the user. The new technology provides greater programmer productivity, better quality of software and lesser maintenance cost. The major characteristic are:

- **Ease in division of job:** Since it is possible to map objects of the problem domain to those objects in the program, the work can be easily partitioned based on objects.
- **Reduce complexity:** Software complexity can be easily managed.
- **Provide extensibility:** Object Oriented systems can be easily upgraded from small to large system.
- **Eliminate redundancy:** Through inheritance we can eliminate redundant code and extend the use of existing classes.

- Saves development time and increases productivity: Instead of writing code from scratch, solutions can be built by using standard working modules.
- Allows building secure programs: Data hiding principle helps programmer to build secure pro grams that cannot be accessed by code in other parts of the program.
- Allows designing simpler interfaces: Message passing techniques between objects allows making simpler interface descriptions with external systems.

1.4 Differences between OOP and Procedural Programming (POP)

Differences between object-oriented programming (OOP) and procedure-oriented programming (POP)

POP	OOP
Emphasis on doing things (procedure). Programs are divided into what are known as functions.	Emphasis on data rather than procedure. Programs are divided into what are known as objects.
Data move openly around the system from function to function.	Data is hidden and cannot be access by external functions.
Employs top down approach in the program design.	Employs bottom up approach in program design.
In POP, importance is given to the sequence of things to be done i.e. algorithms	in OOP, importance is given to the data.
In POP, larger programs are divided into functions	in OOP, larger programs are divided into objects.
In POP, most functions share global data i.e data move freely around the system from function to function	In OOP mostly the data is private and only functions inside the object can access the data.
POP follows a top down approach in problem solving	OOP follows a bottom up approach.
In POP, adding of data and function is difficult	in OOP it is easy.
In POP, there is no access specifier	in OOP there are public, private and protected specifier.

1.5 Introduction to Java

Java is a programming language created by James Gosling from Sun Microsystems in 1991. The first publicly available version of Java (Java 1.0) was released in 1995.Over time new enhanced versions of Java have been released. The current version of Java is Java 1.7 which is also known as Java 7.

From the Java programming language the Java platform evolved. The Java platform allows that the program code is written in other languages than the Java programming language and still runs on the Java virtual machine.

Java Virtual machine

The Java virtual machine (JVM) is a software implementation of a computer that executes programs like a real machine.

The Java virtual machine is written specifically for a specific operating system, e.g. for Linux a special implementation is required as well as for Windows.

Java programs are compiled by the Java compiler into so-called bytecode. The Java virtual machine interprets this bytecode and executes the Java program.

1.6 Features of Java

Java provides many features such as:

- **Simple**

Java was designated to be very simple and easy to learn. The syntax of Java has been kept nearer to C++ so that the usage of Java does not require extensive training programs to be undertaken. However, in Java the infrequently used, complex features of C++ have not been included. Thus, a programmer aware of the various object-oriented concepts can easily develop applications in Java.

- **Object Oriented**

Simply stated, object oriented design is a technique that focuses design on the data and on the interfaces rather than modularization of the functionalities or the tools used to develop them. Java uses object-oriented concepts as for basis for S/W design. Java provides a clean and efficient object-based development platform.

- **Robust**

The multi-platform environment of the web places high demand on the reliability of the program to execute on a variety of systems. Thus, high priority has been given to create robust and highly reliable programs in the design of Java.

It provides extensive compile-time checking followed by a second level of run time checking. The most common problems in programming languages are related to memory management and exception handling.

The memory management model of Java does not allow the creation of pointers. Java has a true array which allows subscript checking to be performed. Thus Java programmers need not worry about freeing or corrupting memory as the programs cannot overwrite the end of a memory buffer. Also Java has automatic garbage collection once the memory is no longer required. Situations like File not found, or division by few, are examples of exceptional condition which are not handled properly by traditional programming.

- **Secure**

Java is designed to be used in networked and distributed environments where security is of paramount importance. Java supports the creation of applications that cannot be invaded from outside. On the other side, Java programs are executed in their own environment and do not go outside these boundaries unless They are authorized to do so. The authentication techniques are based on the public key encryption method.

- **Architecture Neutral**

Java was designed to support applications on heterogeneous network environments composed of a variety of processors, the operating system architectures, and multiple programming language interfaces. To enable a Java application to execute anywhere on the network, the Java compiler generates the bytecode instructions which are not dependent upon a particular computer architecture. These instruction are then interpreted on any machine and translated into the native machine code on the fly by the Java runtime.

- **Portable**

Besides being architecturally neutral, Java is strict in its definition of the basic language. Unlike C or C++, there are no "implementation dependent" aspect of the specification. The size of the primitive data types are specified as is the behaviour of the arithmetic on them. For example, "int" always means a 32 bit integer. Thus, java programs are the same on any platform. There are no data type incompatibilities across the hardware and software architectures.

- **Interpreted**

Java bytecode is not directly executed by the system, because Java is interpreted However, the speed is more than adequate for most interactive application.

- **Multithreaded**

Java was designed to meet the real world of creative, interactive, networked programs. Java's multithreading capability provides the means to build applications with many concurrent threads of activity. The multithreading feature of Java has various sophisticated synchronization primitives. Moreover, Java's high level system libraries have been written to be thread safe, i.e., the functionality provided by the libraries is available without conflict to multiple concurrent threads of execution.

- **Dynamic**

While the Java compiler is strict in its static checking during compile time, the language and run time systems are dynamic during linking and loading stages. Classes are linked only as needed. New code modules can be linked in on demand from a variety of sources, even from sources across the network in a large number.

1.7 Java Application and Applets

The differences between them are as follows:

Applets	Applications
Applets can be embedded in HTML pages and downloaded over the Internet, or Intranet.	Applications have no special support in HTML for embedding or downloading.
Applets can only be executed inside a Java-mand line compatible container, such as a modern Web Browser.	Applications can be executed from the com with a small booting utility such as javac.exe or Java.exe.
Applets execute under strict security limitations that disallow certain operations, such as accessing files or systems services on the user's computer.	Applications have no inherent security restrictions.
Applets are the programs written specially for distribution over a network. These programs contain information to be delivered to the world and involve user interaction, for example, order entry form, registration form mailing, etc.	Applications are system level programs i.e., these programs run in the background and don't involve user interaction, for example, server administration, security manager, etc.

Java offers two flavors of programming, Applets and Applications. They have the following common characteristics:

1) Both programs are composed of one or more files with the "CLASS" extension and having machine independent Java Bytecode.

2) Both require JVM (Java Virtual Machine) to execute these Bytecodes.

1.8 JDK source file structure

Java comes in two flavors, the Java Runtime Environment (JRE) and the Java Development Kit (JDK).

The Java runtime environment (JRE) consists of the JVM and the Java class libraries and contains the necessary functionality to start Java programs.

The JDK contains in addition the development tools necessary to create Java programs. The JDK consists therefore of a Java compiler, the Java virtual machine, and the Java class libraries.

Java Source File (*.java)

Java Compiler (javac)

Java Bytecode File (*.class)

Java Virtual Machine (Java)

Javaap (Java Disassembler)

Java (Java Interpreter)

Appletviewer (for viewing Java applets)

Jdb (Java Debugger)

1.9 Summary

In this introductory unit, you learned about the difference between java applet and java application, java features, and java libraries, etc. Java programs can run from a web browser. Java programs that run from a web page are called java applets. Applets are modern GUIs, including buttons, te xt fields, text area menus, etc. Applets can respond to events such as mouse movements .Unlike other conventional programming language, Java supports many features such as multithreaded, distributed, objected oriented, and robust programming language.

1.10 Self-Assessment Questions

1. List the important Java features.
2. What do you understand by Multithreading?
3. How does Java programming language support robustness?
4. Java is object oriented Programming language. What are its advantages?
5. What is the Java virtual machine (JVM)? Explain.
6. Explain the OOP's concepts.

1.11 References

1. Programming with Java A Primer, E.Balaguruswamy Tata McGraw Hill Companies
2. Java Programming John P. Flynt Thomson 2nd
3. Java Programming Language Ken Arnold Pearson
4. The complete reference JAVA2, Herbert schildt. TMH
5. Big Java, Cay Horstmann 2nd edition, Wiley India Edition
6. Core Java, Dietel and Dietel
7. Java - Balaguruswamy
8. Java server programming, Ivan Bayross SPD

Unit-2

Fundamentals of Java Programming

Structure of the Unit

2.0 Objective

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

- Initialize variables of different data types;
- Use datatypes - their usage and typecasting;
- Use different types of operators and their hierarchy;
- Simple Input and Output program

2.1 Introduction

The building blocks for all objects in Java are the primitive data types in the Java language. In this section, we'll study Java's primitive data types and learn how to name, store, retrieve, assign, and operate on these types. We'll introduce the concept of a variable. A variable provides a convenient way to name an instance of a data type. For a variable to exist, however, it first must be declared. For a variable to be used, it first must have a value assigned to it. To perform meaningful programming tasks with variables, we need a set of operations that can be performed with variables. Strings are not primitive data types in Java. Java includes the following eight primitive data types:

Java has 8 primitive data types. These can be put in four groups:

1. Integers includes byte, short, int, and long
2. Floating-point numbers includes float and double
3. Characters includes char, like letters and numbers.
4. Boolean includes boolean representing true/false values.

2.2 Data Types in Java

Integers and Floating-Point Numbers : There are two general categories of numbers: integers and real numbers. Integers are whole numbers and have no fractional or decimal component. So, 2 is an integer, but 2.718 is not. Many quantities are adequately represented by integers. For example, the number of passengers in a car could be 4 or 5, but not 4.5. Many quantities, however, need a fractional or decimal component, such as the ratio of the circumference of a circle to its diameter, known as pi. For these quantities, a real number is required. Because of the way numbers are stored and represented in computer systems, real numbers are more commonly called floating-point numbers.

The most commontype of integer is int, but Java also supports long integers (long), short integers (short), and byte integers (byte). The most common type of floating-point number is double, for "double-precision", but Java also supports single-precision floating-point numbers (float).

Reverse Word	**Data Type**	**Size**
byte	Byte-length Integer	1 byte
short	Short integer	2 bytes
int	Integer	4 bytes
long	Long integers	8 bytes
float	Single precision 4 bytes number	
double	Real number with double precision	8 bytes
char	Character	1 Bytes

Table 2.1 Range of Primitive Data Types

2.2.1 Characters

Besides numbers, computers must store and manipulate characters, such as letters of the alphabet, digits, punctuation symbols, etc. The primitive data type char serves this purpose. In Java a character is coded as the symbol for the character enclosed in single quotes. Examples include 'A', 'a', '9', '%', and '='.

The syntax to create a character variable is:

Char ch = 'b'; In this example, the 'ch variable has been assigned the value of the letter 'b'. Notice the single quotes around the letter 'b': these tell the compiler that you want the literal value of 'b', rather than an identifier called 'b'.

2.2.2 Escape Sequences

Besides graphic characters that are printable, the char data type can hold values that control input/output devices or modify the appearance or layout of information. These are known as control characters. Since control characters are not printable, they cannot be scripted as a symbol enclosed in single quotes, like other characters. Instead they are represented as escape sequences. An escape sequence is a two-character pattern beginning with the backslash character (\).

Description	Escape Sequence
Line Feed	\n
Carriage Return	\r
Horizontal Tab	\t
Backslash	\\
Single Quote	\'
Double Quote	\\'

Table 2.2. Java Escape Sequences

Boolean Variables: A boolean variable can have only one of two possible values, true or false. This is the type returned by all relational operators (>, < etc.).

2.3 Variables and Constants

We are now ready to put Java's primitive data types to work. A primitive data type may appear in a program two ways: as a variable or as a constant. A constant, sometimes called a literal, is simply an explicit instance of a value. For example, 3 is an integer constant, 3.14 is a floating point constant, 'a' is a character constant, and true is a boolean constant. Integer constants can be expressed in hexadecimal notation, if desired. For example 0x00FF is an integer constant equal to 255. The pattern "0x" means that what follows is interpreted in hexadecimal notation.

A variable is an abstraction that represents a value. A key difference between a constant and a variable is that a constant, once coded in a program, cannot change. A variable holds a value, but this value can change as a program executes.

Before a variable is used, it must be declared and it must be initialized to a value. The following four Java statements declare integer, floating-point, character, and boolean variables:

```
int count;

double temperature;

char c;

boolean status;
```

Each line above is an example of a Java statement. Note that a statement ends with a semicolon. In fact, each line above is a particular type of Java statement known as a declaration. The first line declares a variable named count of type int. The second line declares a variable named temperature of type double. And so on. The variable's name is chosen by the programmer.

The following eight program statements show the declaration of variables followed immediately by the initialization of the

variables:

```
// declare four variables

int count;

double temperature;

char c;

boolean status;
```

// initialize the four variables to values

count = 7;

temperature = 98.6;

c = 'A';

status = true;

2.4 Comments

Comments are notations included in a source program to help explain an operation. There are two ways to include comments in Java programs. When two forward slash characters (//) appear on a line, the rest of the line is a comment and is ignored by the compiler. This technique is useful for short comments, such as a one-line comment or a comment appended to the end of a program statement. Long comments, perhaps spanning many lines of source code, can use "/*" and "*/" to delimit the comment, such as

/* this is a comment */

Comments that span many lines are usually called a comment block.

2.5 Identifiers and Reserved Words

Whenever a programmer creates a name for a variable or other component of a program (e.g., the name of a class), the result is an identifier. So, in the preceding examples, temperature was an identifier, status was an identifier, and so on. Clearly, there must be rules on whatconstitutes a legal or illegal identifier. And there are. A Java identifier can consist of any letters or digits, as well as an underscore (_), or a dollar sign ($). A digit character, however, cannot be the first character in an identifier. The following are examples of legal identifiers:

number_of_items

inputMode

X99

temp$filename$suffix

The following are examples of illegal identifiers:

99gretzky (Illegal: can't start with a digit)

odd-ball (Illegal: can't use a dash)

this.that (Illegal: can't use a period)

There is no size restriction on identifiers, so if you want an identifier with a hundred characters,

you can have it. Java is case sensitive, however, so beware that the following two identifiers are

different:

pressure

Pressure

As a programmer, you can make-up identifiers that are as cryptic or as meaningful as you wish. Obviously, well-chosen identifiers help to clarify program statements. Besides the few rules noted above, an identifier must not conflict with Java's reserved words. Think of the problems if an integer variable were named float. The statements int float; // wrong!

float = 3; // wrong!

would make the compiler's job quite difficult. Java's 59 reserved words are:-

abstract	**boolean**	**break**	**byte**	**byvalue**
case	cast	catch	char	class
const	continue	default	do	double
else	extends	false	final	finally
float	for	future	generic	goto
if	implements	import	inner	instanceof
int	interface	long	native	new
null	operator	outer	package	private
protected	public	rest	return	short
static	super	switch	synchronized	this
throw	throws	transient	true	try
var	void	volatile	while	

Figure 2.1 :Java's reserved words

Naming Conventions

Identifiers are used for more than just the names of variables. They are also used for the names of methods, classes, and defined data constants. For each of these, naming conventions exist, and adhering to these will contribute to making your code understandable.

Type of identifier	**Example**	**Convention**
Variable	temperature moredata	starts with lowercase character multiple words joined together first letter of words (except first) in uppercase

Method	length()
	tolowerCase()
		same rules as for variables
		·parentheses distinguish methods from variables
		exception: constructor methods begin with an uppercase character

class	String MenuBar	starts with uppercase character multiple words joined together first letter of words in uppercase
Constant	FACTOR SCREEN_WORTH	all characters in uppercase multiple words joined with underscore character

Figure 2.3 Conventions for naming identifiers

Literals

Literals are pieces of Java source code that indicate explicit values. For instance, Hello World! Is a String literal. Java has four kinds of literals: String, Character, Number, and Boolean.

String Literals

The string literal is always enclosed in double quotes. Java uses a String class to implement strings, whereas C and C++ use an array of characters. For example:

"Hello World!"

String message = "Hello World";

Character Literals

Character literals are similar to String literals except they are enclosed in single quotes and must have exactly one character. For example 'c' is a character literal. A backslash is used to denote the non-printing characters such as

Boolean Literal

A Boolean literal can have either of the values: true or false. They do not correspond to the numeric values, 1 and 0, as in C and C++.

Numeric Literals

There are two types of numeric literal: integers and floating point numbers. For example: 34 is an integer literal, and it means the number thirty four.

1.5 is a floating point literal

45.6, 76.4E8 (76.4 times 10 to the 8th power) and 32.0 are also floating point literals.

Variables only exist within the structure in which they are defined. For example, if a variable is created within a method, it cannot be accessed outside the method. In addition, a different method can create a variable of the same name which will not conflict with the other variable. A java variable can be thought of as a little box made up of one or more bytes that can hold a value of a particular data type:

Syntax: variabletypevariablename = data;

Source Code (demonstrating declaration of a variable)

```
class example
{
public static void main ( String[] args )
  {
long x = 123;    //a declaration of a variable named x with a datatype of long
System.out.println("The variable x has: " + x );
  }
}
```

2.6 Operators in Java

Representing and storing primitive data types is, of course, essential for any computer language. But, so, too, is the ability to perform operations on data. Java supports a comprehensive set ofoperators to facilitate adding, subtracting, multiplying, dividing, comparing, etc. Let's start with a simple demonstration program. Following is the Java program that adds one and one to get two.

```
public class Numbers
{
        public static void main(String[] args)
        {
int x;
                x = 1;
int y = 1;
                 int z = x + y;
                System.out.print("1 + 1 = ");
                System.out.println(z);
        }
        }
```

Program : Numbers.java

When this program executes, it generates the following output:

```
1 + 1 = 2
```

In line 5, an int variable named x is declared. The effect is to set aside storage for x, but the storage is not initialized with a value (see Figure 2a). In line 6, the variable x is assigned thevalue 1. Theequals sign (=) is known as the assignment operator. The effect is to place the value1 into the storage location set aside for the variable x (see Figure 2b).

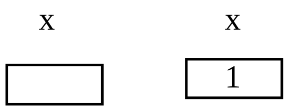


Figure 2. (a) A variable x is declared, but not initialized blank(b) a variable x is initialized with the value 1.

In line 7, we see declaration combined with assignment. A new int variable, y, is declared and initialized with the value 1.Line 8 contains another example of declaration combined with assignment. An int variable z is declared and assigned the value x + y. The plus sign (+) is the addition operator. Since line8 contains two operators, there is a need to process the operations in a certain order. This process isgoverned by Java's precedence of operators. As you might guess, addition takes precedence overassignment; so, the expression x + y is evaluated first and then the result is assigned to z. The result is printed in lines 9 and 10. Java also includes operators for subtraction (-), multiplication (*), and division (/). Where these and other operators are mixed, precedence must be carefully considered. Let's examine precedence of operators in more detail. The program Operators.java shows subtraction and multiplication combined in a single ex-pression.

```
        public class Operators
        {
           public static void main(String[] args)
            {
              inti=  12 - 5  * 3;
              int j = (12 - 5) * 3;
             System.out.println(i);
  System.out.println(j);
```

```
        }
    }
```

Figure Operators.java

The output of this program is

```
-3
21
```

In line 5, an int variable i is declared and assigned the result of the expression 12 - 5 * 3. If weevaluate the expression left-to-right the answer is 21, which is wrong because multiplicationtakes precedence over subtraction. The expression 5 * 3 is evaluated first and the result, 15, is then subtracted from 12 to yield the correct answer of -3. To perform the subtraction first, ifdesired, parentheses are added as shown in line 6. Parentheses override the natural precedence of operators.

Division and multiplication are of equal precedence, as are addition and subtraction. Whenarithmetic operators of the same precedence appear together, they are evaluated left to right. So, the expression 2 + 3 - 5 - 6 equals -6 and the expression 30 * 4 / 3 / 8 equals 5.

The Remainder Operator

When dividing integers, the decimal portion of the result, or the remainder, if any, is lost. So, theexpression 13 / 5 equals 2. The remainder after division of integers can be obtained using the remainder (%) operator. (The remainder operator is sometime called the modulus or modoperator). So, the expression 13 % 5 equals 3, because 13 divided by 5 is 2 with a remainder of3. The program Days demonstrates the mod operator in action.

```
1  public class Days
2 {
3   public static void main(String[] args)
4     {
5     int seconds =  1000000; //one million
6
7       int days      = seconds / (24 * 60 * 60);
8       int remainder = seconds % (24 * 60 * 60);
9       int hours     = remainder / (60 * 60);
10      remainder     = remainder % (60 * 60);
11      int minutes   = remainder /  60;
12      remainder     = remainder %  60;
13
14     System.out.println(seconds   + " seconds = ");
15     System.out.println(days      + " days");
16    System.out.println(hours    + " hours");
17            System.out.println(minutes  + " minutes, and");
18          System.out.println(remainder + " seconds");
19            }
20        }
```

Figure Days.java

The Days.java program computes the number of days, hours, minutes, and seconds in onemillion seconds. The output is

```
1000000 seconds =
11 days
13 hours
46 minutes, and
40 seconds
```

In line 7, the number of days in one million seconds is computed by dividing the int variablesecondsby (24 * 60 * 60). The result is 11. The arithmetic is for integers, so the remainder is lost. The remainder is retrieved in line 8 by performing the same operation again, except usingthe mod operator. The result - the remainder - is assigned to the int variable remainder.

The number of hours is computed by dividing remainder by (60 * 60) in line 9. A similarprocess is repeated for hours, minutes, and seconds (lines 10-12).When one or both values are negative, extra caution is warranted. The operations must obey the following rule:

(x / y) * y + x % y == x

So, for example

 7 % 2 equals 1
 7 % -2 equals 1
-7 % 2 equals -1, and
-7 % -2 equals -1.

Although much less used, the remainder operator also works with floating-point numbers. In this case, it returns the remainder after "even" division, for example, the expression 8.0 % 2.5returns 0.5.

The % operator has the same precedence as multiplication and division. Precedence of OperatorsAlthough Java includes many more operators, let's summarize what we have learned thus farabout the precedence of operators (see Table 2.4)

Precedence	**Operator(s)**	**Operation**	**Association**
highest	( )	Overridenatural precedence	Inner to outer
next highest	* / %	multiplication, division, remainder	L to R
next highest lowest	+ - =	addition, subtraction assignment	L to R R to L

Table 2.4 Precedence of Operators

The association is left-to-right for most operators. One exception is the assignment operator. In the event of more than one assignment operator in a single statement, the association is right toleft. Consider, for example, the following statements:

```
int x = 2;

int y = 3;

int z;

int a = z = x + y;
```

In final line, the expression x + y is evaluated first, then the result is assigned to z. Finally, thevalue of z is assigned to a.

The cast Operator

The cast operator explicitly converts one data type to another. Casting is useful, for example,where a conversion is necessary that does not automatically occur through promotion. Casting is also used to convert one object type to another, as we will meet later. The syntax for the castoperator is

(type)variable

or

(type)constant

The cast operator consists of the name of a data type enclosed in parentheses followed by avariable or constant. As a simple example, the following statements

```
double x = 3.14;

int y = (int)x;

System.out.println(y);
```

print 3 on the standard output. In the second line, the value of a double variable x is assigned to the int variable y. This is permitted because the double is cast to an int by preceding x with "(int)". Note that a floating-point number is truncated (not rounded) when cast to anint.

At this point, you might be asking yourself, "Why is the conversion from one data type to another automatic in some cases, but only allowed through casting in other cases?" Anytime there is apotential "loss of information", casting is required. The cast operator is like a safety check; it isyour way of telling the compiler that you're aware of the possible consequences of the conversion.

Let's explore casting and promotion in more detail through an example program. Figure contains the listing for DemoPromoteAndCast.java.

```
public class DemoPromoteAndCast

{

  public static void main(String[] args)

   {

    // integer promotion

      byte  a = 1;

     short b = a;          // byte promotes to short

     int   c = b;          // short promotes to int
```

```
        long  d = c;         // int promotes to long
       // integer casting
        c = (int)d;          // long casted to int
        b = (short)c;         // int casted to short
        a = (byte)b;          // short casted to byte
       // floating-point promotion
        float  x = 1.0f;      // 'f' needed to indicate float
        double y = x;         // float promotes to double
       // character example
         char aa = 'Q';
        int  xx = aa;         // character promotes to int
        char bb = (char)xx;    // int casted to char
       // floating-point casting
        x = (float)y;         // double casted to float
        c = (int)y;           // double casted to int
      }
    }
```

Figure DemoPromoteAndCast.java

This program is as dull as they get. It receives no input and it generates no output. The key point is that it compiles without errors. All statements are perfectly legal Java statements. The objective is to exercise the rules of the Java language with respect to promotion and casting. The program is divided into five sections. In lines 5-9, integer promotion is demonstrated,working from "lowest" to "highest". A byte variable, a, is declared in line 6 and assigned thevalue 1. In line 7, a is assigned to the short variable b. Since a byte is 8 bits and a short is 16 bits, a byte is considered a "lower" form of an integer. The assignment is legal because the lower form is automatically promoted to the higher form. A similar process is demonstrated inline 8 (promoting a short to an int) and line 9 (promoting an int to a long).

Integer casting is demonstrated in lines 11-14. Now we are working down the hierarchy ofinteger types, and this requires casting. Assigning a long to an int is permitted only if thelong is cast to an int, as demonstrated in line 12. Similarly, an int can be assigned to ashort by casting (line 13), and a short can be assigned to a byte by casting (line 14).

Java contains only two variations of floating-point numbers, float and double, and float (32 bits) is "lower" than double (64 bits). Assigning a float to a double is allowed (line 18), as automatic promotion occurs. In line 17, note that the constant assigned to the floatvariable x is coded as 1.0f. A trailing 'f' or 'F' indicates a floating-point number isintended as a float. Appending 'd' or 'D' indicates double, but this is never needed since double is the default for floating-point numbers.

A character example appears in lines 20-23. A char variable aa is declared and assigned theconstant 'Q'. In line 23, the char variable aa is assigned to the int variable xx. This ispermissible as the char is automatically promoted to an int. The value assigned is the Unicodevalue of the character.

Finally, floating-point casting is demonstrated in lines 25-27. First, a double is assigned to afloat (line 26). Since float is the lower of the two types, casting is required. Finally, adouble is assigned to anint through casting.

Increment, Decrement, Prefix, Postfix

Adding or subtracting one (1) is so common in Java and other programming languages that ashorthand notation has evolved. For simple increment by 1 the ++ operator is used, and forsimple decrement by 1, the -- operator is used. So, the following two lines have the same effect:x = x + 1;

x++;

as do

x = x - 1;

x--;

When these operators appear after the variable, as above, they are known as postfix operators. They can also appear in front of a variable as prefix operators. Although in both cases the effect is to increment or decrement the variable, there is a small but important difference. This difference is illustrated as follows:

int x = 5;

int y = 3 + x++;

After these statements execute, y equals 8 and x equals 6. Look carefully and see if you agree. Here's the operation: If a variable in an expression is bound with a postfix increment operator, the value used is the original value of the variable. The variable is incremented after the value isused in the expression. A similar statement can be made for the decrement operator. For theseoperators in the prefix position, the increment or decrement occurs before the variable is used in the expression. Prefix and postfix operators are summarized in Table 2.5.

Expression	Effect	Value of Expression
x++	Increment x by 1	x
++x	increment x by 1	x+1
x--	decrement x by 1	x
--x	decrement x by 1	x - 1

Table 2.5 : Prefix and Postfix Operators

Operation-Assignment Shorthand

A popular shorthand notation has evolved for operators that normally take two arguments - one on the left, one on the right - such as the arithmetic operators. When these operators arecombined with assignment, they can be coded using op= notation. For example, the followingtwo statements have the same effect:

x = x + 3;

x += 3;

as do

z = z / y;

z /= y;

2.7 Simple Program Input and Output

In this section we will show how to receive input from the keyboard and process that input within a Java program.

The most common user interface devices are the CRT display for output and the keyboard for input. Just as System.out was used to output data on the host system's CRT display,System.in is used to receive input from the keyboard. To do this, however, the following initialization is required:

BufferedReaderstdin = new BufferedReader(new InputStreamReader(System.in), 1);

This statement sets-up System.in as a buffered character input stream. There are two important services provided. The first is to convert raw bytes arriving from the keyboard into characters. This service is performed by the InputStreamReader class. Second, the characters are buffered to provide efficient reading from the input stream. Thisservice is performed by the BufferedReader class. The statement declares and instantiates an object named stdin of the BufferedReader class. This is analogous to earlier statements that declared and initialized variables. Now, however, instead of a primitive data type, we have a class (BufferedReader) and instead of a variable, we have an object (stdin).

Following the above statement, a line of text is read from the keyboard as follows:

String line = stdin.readLine();

Think of stdin.readLine() as an expression, which it is. As an expression, it is evaluated and it yields a value. This value is assigned to the variable line. More precisely, the readLine() method is called on the stdin object - the keyboard. A string is returned and assigned to the String variable line. The string contains the characters of a line of text entered on the keyboard.

A line of input ends when the user presses the Enter key. The Enter key generates an end-of-line code, but this is not included in the string returned by the readLine() method. The string can be printed:

System.out.println(line);

Figure shows the listing for a program that prompts the user to enter a name, age, and the radius of circle.

```
import java.io.*;
public class DemoKeyboardInput
{
  public static void main(String[] args) throws IOException
   {
    // open keyboard for input (call it 'stdin')
    BufferedReaderstdin =
      new BufferedReader(new InputStreamReader(System.in), 1);
             // get input from the keyboard
    System.out.print("Please enter your name: ");
   String name = stdin.readLine();
     System.out.print("Please enter your age: ");
    String s1 = stdin.readLine();
     System.out.print("Please enter the radius of a circle: ");
```

```
                  String s2 = stdin.readLine();
                   // perform conversions on input strings
                  int age = Integer.parseInt(s1);        // string to int
                  double radius = Double.parseDouble(s2);  // string to double
                     // operate on data
                   age++;                              // increment age
                  double area = Math.PI * radius * radius; // compute circle area
                    // output results
                  System.out.println("Hello " + name);
                  System.out.println("On your next birthday, you will be "
                      + age + " years old");
                  System.out.println("Area of circle is " + area);
                 }
              }
```

Figure DemoKeyboardInput.java

A sample dialogue with DemoKeyboardInput follows: (User input is underlined.)

```
PROMPT>java DemoKeyboardInput
Please enter your name: Nimit
Please enter your age: 18
Please enter the radius of a circle: 6.7
Hello Nimit
On your next birthday, you will be 19 years old
Area of circle is 141.02609421964584
```

The program HeightConversion prompts the user to enter a height in feet and inches. The output is the height in centimeters.

```
import java.io.*;
             public class HeightConversion
             {
    private static final int INCHES_PER_FOOT = 12;  // inches per foot
              private static final double FACTOR = 2.564;    // cm per inch
              public static void main(String[] args) throws IOException
               {
                BufferedReaderstdin =
                  new BufferedReader(new InputStreamReader(System.in), 1);
       System.out.println("Please enter your height in feet and inches");
                System.out.print("Feet: ");
                int feet = Integer.parseInt(stdin.readLine());
                System.out.print("Inches: ");
                double inches = Double.parseDouble(stdin.readLine());
                 double cm = (feet * INCHES_PER_FOOT + inches) * FACTOR;
                System.out.print("Height = " + cm + " cm");
               }
  }
```

Figure HeightConversion.java

A sample dialogue follows:

```
PROMPT>java HeightConversion
Please enter your height in feet and inches
Feet: 5
Inches: 7.5
Height = 173.07 cm
```

? This figure shows how to calculate area of Rectangle using it's length and width.

```
importjava.io.BufferedReader;
importjava.io.IOException;
importjava.io.InputStreamReader;
public class CalculateRectArea {
public static void main(String[] args) {
int width = 0;
int length = 0;
              //read the length from console
          BufferedReaderbr = new BufferedReader(newInputStreamReader(System.in));
          System.out.println("Please enter length of a rectangle");
length = Integer.parseInt(br.readLine());
                //read the width from console
          System.out.println("Please enter width of a rectangle");
          width = Integer.parseInt(br.readLine());
          int area = length * width;
          System.out.println("Area of a rectangle is " + area);
                }
          }
```

Output of Calculate Rectangle Area using Java Example would be

```
Please enter length of a rectangle
10
```

```
Please enter width of a rectangle
15
Area of a rectangle is 150
```

2.8 Decision Making Statement in Java

There are two types of decision making statements in Java. They are:

if statements

switch statements

The if Statement: An if statement consists of a Boolean expression followed by one or more statements.

Syntax:

The syntax of an if statement is:

```
if(Boolean_expression)
{
  //Statements will execute if the Boolean expression is true
}
```

If the boolean expression evaluates to true then the block of code inside the if statement will be executed. If not the first set of code after the end of the if statement(after the closing curly brace) will be executed.

Example:

```
public class Test {
public static void main(String args[]){
int x = 10;
if( x < 20 ){
System.out.print("This is if statement");
}
}
}
```

This would produce following result:

```
This is if statement
```

The if...else Statement: An if statement can be followed by an optional else statement, which executes when the Boolean expression is false.

Syntax:

The syntax of a if...else is:

```
if(Boolean_expression){
  //Executes when the Boolean expression is true
}else{
  //Executes when the Boolean expression is false
}
```

Example:

```
        public class Test {
        public static void main(String args[]){
        int x = 30;
        if( x < 20 ){
        System.out.print("This is if statement");
        }else{
        System.out.print("This is else statement");
        }
        }
        }
```

This would produce following result:

```
This is else statement
```

The if...else if...else Statement: An if statement can be followed by an optional else if...else statement, which is very usefull to test various conditions using single if...else if statement. When using if , else if , else statements there are few points to keep in mind. An if can have zero or one else's and it must come after any else if's. An if can have zero to many else if's and they must come before the else. Once an else if succeeds, none of he remaining else if's or else's will be tested.

Syntax:

The syntax of a if...else is:

```
if(Boolean_expression 1){
  //Executes when the Boolean expression 1 is true
}else if(Boolean_expression 2){
  //Executes when the Boolean expression 2 is true
}else if(Boolean_expression 3){
  //Executes when the Boolean expression 3 is true
}else {
  //Executes when the none of the above condition is true.
}
```

Example:

```
        public class Test {
        public static void main(String args[]){
        int x = 30;
```

```
if( x == 10 ){
System.out.print("Value of X is 10");
}else if( x == 20 ){
System.out.print("Value of X is 20");
}else if( x == 30 ){
System.out.print("Value of X is 30");
}else{
System.out.print("This is else statement");
}
}
}
```

This would produce following result:

```
Value of X is 30
```

Nested if...else Statement:

It is always legal to nest if-else statements, which means you can use one if or else if statement inside another if or else if statement.

Syntax:

The syntax for a nested if...else is as follows:

```
if(Boolean_expression 1){
  //Executes when the Boolean expression 1 is true
if(Boolean_expression 2){
    //Executes when the Boolean expression 2 is true
  }
}
```

You can nest else if...else in the similar way as we have nested if statement.

Example:

```
public class Test {
public static void main(String args[]){
int x = 30;
int y = 10;
if( x == 30 ){
if( y == 10 ){
System.out.print("X = 30 and Y = 10");
}
}
}
```

This would produce following result:

```
X = 30 and Y = 10
```

The switch Statement:

A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax:

The syntax of enhanced for loop is:

```
switch(expression){
case value :
    //Statements
break; //optional
case value :
    //Statements
break; //optional
  //You can have any number of case statements.
default : //Optional
    //Statements
}
```

The following rules apply to a switch statement:

The variable used in a switch statement can only be a byte, short, int, or char. You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon. The value for a case must be the same data type as the variable in the switch, and it must be a constant or a literal. When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached. When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.

Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached. A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Example:

```
public class Test {
public static void main(String args[]){
char grade = args[0].charAt(0);
switch(grade)
{
case 'A' :
System.out.println("Excellent!");
break;
case 'B' :
case 'C' :
System.out.println("Well done");
break;
case 'D' :
System.out.println("You passed");
case 'F' :
System.out.println("Better try again");
break;
default :
System.out.println("Invalid grade");
}
System.out.println("Your grade is " + grade);
}
}
```

Compile and run above program using various command line arguments. This would produce following result:

```
$ java Test a
Invalid grade
Your grade is a
$ java Test A
Excellent!
Your grade is A
$ java Test C
Well done
Your grade is  C
$
```

2.9 Summary

The building blocks for all objects in Java are the primitive data types in the Java language. In this chapter , Java's primitive data types are discussed and you also learn how to name, store, retrieve, assign, and operate on these types. You also understand the concept of a variable. A variable provides a convenient way to name an instance of a data type. For a variable to exist, however, it first must be declared. For a variable to be used, it first must have a value assigned to it. To perform meaningful programming tasks with variables, we need a set of operations that can be performed with variables. We have discussed in this chapter , a brief about Java Variables, Literal, Constant and use of data types with hierarchy of operators . The chapter also define simple input and output program with decision making statement in java.

2.10 Self-Assessment Questions

1. Write short note on Data-types of Java
2. Explain the terms Variable, Constant, Literal use in java
3. Define Decision Making Statement use in Java Programming
4. Write a program in Java for finding biggest between two number
5. Explain the term Switch in Java

2.11 References

1. Programming with Java A Primer, E.Balaguruswamy Tata McGraw Hill Companies
2. Java Programming John P. Flynt Thomson 2nd
3. Java Programming Language Ken Arnold Pearson
4. The complete reference JAVA2, Herbert schildt. TMH
5. Big Java, Cay Horstmann 2nd edition, Wiley India Edition
6. Core Java, Dietel and Dietel
7. Java - Balaguruswamy
8. Java server programming, Ivan Bayross SPD

Unit-3
Operators in Java

Structure of the Unit

3.0 Objective

The objective of this unit is to learn

- logical (Boolean) operators
- use of orders of precedence,
- selection structures in Java.
- built-in Java methods,
- user define methods

3.1 Relational Expressions

We have used the term expression quite a bit in the preceding notes. For the most part, the idea is straightforward. An expression is simply one or more variables and/or constants joined byoperators. An expression is evaluated and produces a result. The result of all expressions thus far was either an integer value or a floating-point value.

But, an expression can also yield a boolean, a result that is either true or false. Such anexpression is called a relational expression. The result reflects how something "relates to"something else. For example, "Is the value of x greater than the value of y?" Note that thepreceding poses a question. Relational expressions are usually intended to answer yes/no, ortrue/false, questions. Obviously, boolean values and boolean variables play an importantrole in relational expressions.

3.2 Relational Operators

To build relational expressions, two types of operators are used: relational operators and logicaloperators. Let's deal first with the relational operators. There are six relational operators, four of equal precedence:

>	greater than
>=	greater than or equal to
<	less than

<= less than or equal to

and two just below these in precedence:

== equal to

!= not equal to

These operators, like the arithmetic operators met earlier, are sometimes called binary operators,because they take two arguments - one on each side. The arguments are generally integers orfloating-point variables or constants. The result is a boolean. The following are examples of relational expressionsbuilt from relational operators: (the result is also shown)

3 < 4	true
7.6 <= 9	true
4 == 7	false
8.3 != 2.1	true

In the second line, a double is compared with an int. The int is promoted to doublebefore the expression is evaluated.So, what can you do with a relational expression? Since a relational expression yieldseithertrue or false, the result can be assigned to a boolean variable. Furthermore, a Boolean variable, like an int or double, can be printed. So, the following two statementsboolean b = 8.3 != 2.1;

System.out.println(b);

print "true" on the standard output. A much more common use of relational expressions is tocontrol program flow in if statements or in while, do/while, or for loops. relational expressions can also employ variables, as in the following sequence of Java

statements:

```
int x = 3;
int y = 4;
boolean b = x > y;
System.out.println(b);
```

In the third line, the relational expression x > y is evaluated and the result, false, is assigned to the boolean variable b. The fourth statement prints "false".

3.3 Logical Operators

There are three logical operators:

&& AND (true if both arguments are true, false otherwise)

|| OR (true if either argument is true, false otherwise)

! NOT (true if argument is false, false otherwise)

The descriptions in parentheses are usually laid out in truth tables, as shown in following tables

a	b	a &&b
false	false	False
false	true	False
true	false	False
true	true	True

Table 3.1 Truth tables for logical operators. (a) AND

a	b	a || b
false	false	False
false	true	True
true	false	True
true	true	True

Table 3.2 Truth tables for logical operators. (b) OR

a	!a
false	true
true	false

Table 3.3 Truth tables for logical operators. (c) NOT

Logical operators are similar to relational operators in that they both produce boolean results. However, they differ in that logical operators also use boolean arguments. So the following statements make sense:

boolean a = true;

boolean b = false;

boolean c = a && b;

whereas the third statement below has no meaning:

int a = 3;

int b = 4;

boolean c = a && b; // wrong! && requires boolean arguments

The following statements, however, are perfectly reasonable:

int a = 3;

int b = 4;

int c = 5;

int d = 6;

boolean e = a < b && c < d;

System.out.println(e);

and will print "true" on the standard output. This result is by no means obvious, until weacknowledge the precedence relationship between the relational and logical operators. Therelational operators are of higher precedence than the logical AND and logical OR operators. Theparentheses in the following statement illustrate this (although they are not necessary) boolean e = (a < b) && (c < d);

The expression a < b is true (because 3 is less than 4) and the expression c < d is true(because 5 is less than 6). The && (AND) operator yields true if both operands are true, which they are in this example. So, the expression (a < b) && (c < d) is true, and this result is assigned to the boolean variable e.

The && (AND) and || (OR) operators are binary operators, because they take two arguments.

The ! (NOT) operator is a unary operator, because it takes one argument. It returns the logicalcomplement of its argument. So, the following statements

boolean b = true;

System.out.println(!b);

print "false" on the standard output. Note the !operator does not change the value of theboolean variable b; it simply returns a value which is the logical complement of b. Unaryoperators, in general, bind very tightly to their arguments; and the unary ! operator is noexception. It is of higher precedence than all the logical or relational operators.

3.4 Precedence of Operators

This is a good time to re-visit a topic discussed earlier, operator precedence. Table lists all the operators in Java. Most of the operators in the table have been discussed previously. The others will be presented later on an as-needed basis. Each row represents a different precedence level,with operators farther up the table having higher precedence than operators farther down.

Precedence	Operator(s)	Operation
Highest	[] . () expr++ expr--	postfix
	++expr --expr +expr-expr	unary
	new(type)expr	creation or cast
	* / %	Multiplicative
	+ -	Additive
	<<>>>>>	Shift
	><>= <=	Instanceof relational
	== !=	Equality
	&	bitwise AND
	^	bitwise exclusive OR
	|	bitwise inclusive OR
	&&	logical AND
	||	logical OR
	?:	Conditional
lowest	= op=	assignment

Table 3.4: Precedence of Operators (complete)

All binary operators - those receiving two arguments - are left-associative, meaning they areexecuted left-to-right. In other words,4 + 5 - 6 + 7is the same as((4 + 5) - 6) + 7

Of course, if binary operators from different rows in Table are mixed in an expression, then the position in the table determines the order of evaluation. So,5 - 6 * 3is the same as5 - (6 * 3)

Note that the logical AND operator (&&) is of higher precedence than the logical OR (||) operator. So,a&& b || c && dis the same as(a && b) || (c && d)

The assignment operator (=) is right-associative. So,

a = b = cis the same asa = (b = c)

It is common in Java to mix assignment with a boolean test, for example

```
String s;
if ((s = stdin.readLine()) != null)
  /* process line */
```

Since the assignment operator (=) is of lower precedence that the inequality operator (!=), anextra set of parentheses is needed. Reworking the above fragment as

```
String s;
if (s = stdin.readLine() != null)   // WRONG!
  /* process line */
```

results in a compiler error.

The op= entry along the bottom row in Table implies any of

+= -= *= /= %= >>= <<= >>>= &= ^= |=

Bear in mind that operator precedence combined with associativity determines the order ofevaluation. So, with an expression such as

a + b + c

the compiler first evaluates a, then evaluates b, then adds the values of a and b, then evaluates c, then adds the value of c to the previous result.

3.5 Organization of Java

With a firm grounding in the elements of the Java language and in designing programs thatinclude choices and loops, we are well equipped to put Java to work in solving interestingproblems. In this section, we will move further along in our study of Java, as we learn to useadditional classes in Java's Application Programming Interface (API).

The Java API is a collection of packages. Each package contains a collection of classes, and each class contains a collection of methods. Through methods, objects are created and acted upon. In the following sections, we will describe the organization of Java, as seen through its packages,classes, methods, and objects.

What is a Package?

A package is a collection of classes. Packages are simply a convenient place for developers toput classes of a common purpose; however, they play no specific role in the Java language. TheJava developers at Sun Microsystems have organized the many hundreds of classes in the JavaAPI into just over 50 core packages. Within these, hundreds of classes and thousands of methods are defined.

The packages from the Java API encountered in these notes are summarized in Table .

Package	Description
java.applet	provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context
java.awt	contains all of the classes for creating user interfaces and for painting graphics and images ("awt" is an acronym for "advanced windowing toolkit")
java.io	provides for system input and output through data streams, serialization,and the file system
java.lang	provides classes that are fundamental to the design of the Java programming language
java.util	contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization,and miscellane ous utility classes (a string tokenizer, a random-number generator, and a bit array)
javax.swing	provides a set of "lightweight" (all-Java language) components that, to the maximum degree possible, work the same on all platforms

Table 3.5 Packages in the Java API

The import Statement

To use classes from a package in the Java API, the import statement is required at the top of theprogram. For example, the BufferedReader class appears in the java.io package. Any program that uses this class must include the following statement at the top of the source file:import java.io.BufferedReader;

The import statement informs the compiler of the whereabouts of classes used in a programthat are not defined in the program. It is common and convenient to simplify the statement above as follows:

import java.io.*;

The asterisk (*) is a wildcard character. With this statement, any class from the java.iopackage can be used in the program.

The java.lang package is special. It provides classes that are fundamental to the design of theJava language. The most important classes are Object, which is the root of the class hierarchy,and Class, instances of which represent classes at run time. In most cases, we are not aware of the presence or use of these classes, as they operate "behind the scenes" in a program. As well,the java.lang package provides the wrapper classes, the String class, and the Systemclass. Since the java.lang package is so fundamental to the language, the following statement isimplied for all Java programs:

importjava.lang.*; // Implied! Not necessary

What is a Class?

A class is like a data type. Just as we create and operate on variables of a certain data type, we create and operate on objects of a certain class. However, a class is not a primitive data type, likeintor double, because the data it encapsulates are more sophisticated. Central to thissophistication are the methods of each class. So, for example, there is a class named "String",and we can use methods of the String class to perform useful operations on String objects.

However, not all classes fit the description above. Each of our demo programs contained a classbearing the same name as the filename. Clearly, these classes are in no way a "data type". Aswell, Java's Math class is unlike a data type. One cannot create a Math object (like we create aString object). The Math class is simply a convenient holding place for useful methods (toperform mathematical operations) and useful data fields (like the constant PI).

One feature that is common to all classes is that they contain methods. From the abovediscussion, it appears there are two broad categories of classes: those for which objects can becreated, and those for which objects cannot be created. The String class is an example of the former, the Math class, the latter.

What is an Object?

An object is an instance of a class. The act of "instantiating an object" is the act of creating an object of a class and initializing its data fields with values. So, a String object is an instance of the String class. Once an object is instantiated, we can perform operations on it. However,unlike a variable representing a primitive data type, we cannot, in general, directly access the data fields of an object. Our only means to do so is through the methods of that object's class.

What is a Method?

A method is a set of instructions to do something. So, in a programming sense, methods do the work, and objects are the things methods work on. We can categorize all Java methods as being constructor methods, instance methods, or class methods. Let's look at each of these.

Constructor Methods

The purpose of a constructor method is to construct an object. This is often expressed in moreformal terms: The purpose of a constructor method is to instantiate an object. It seemsreasonable from our earlier discussion that the Math class does not have a constructor method(One cannot instantiate a Math object!), whereas the String class does. In fact, this is the case. There is no method called Math(). However, there is a method called String(). It is a rule of the Java language that a constructor method has the same name as its class. We have used a few constructor methods already, such as String(), BufferedReader(), andInputStreamReader(). These are constructor methods for the classes String, BufferedReader, and InputStreamReader, respectively.

A characteristic of Java and other object-oriented programming languages is the ability to havemore than one method with the same name. We saw examples earlier of two methods calledString(). One was called without an argument inside the parentheses, the other was calledwith one argument - a string constant. These are two different methods. The compilerdetermines which to use by context. If the compiler finds a String() constructor with noargument, the default constructor is used. If one argument appears and it is a string constant, then the method with one string constant is used. When two methods have the same name, themethods are said to be overloaded.

Instance Methods and Class Methods

Distinguishing between instance methods and class methods is a bit tricky. Most of the methodsused in demo programs thus far were instance methods. Consider the following two statements:

String s = new String("hello");

intlen = s.length();

There are two methods above: a constructor method named String() and an instance methodnamed length(). Both are methods of the String class. The length() method is aninstance method because it is called through an "instance" of the String class (in this case, s).

Dot notation connects the implicit variable's name to the method's name. So, if an object variable precedes the method's name, the method is an instance method: It is called through an instance of the class.

Now consider the following two statements:

double x = 99.0;

double y = Math.sqrt(x);

The method sqrt() is a class method. (Class methods are also called static methods.) Preceding the method's name is "Math." which is the name of a class. So, what precedes themethod's name helps distinguish a class method from an instance method.

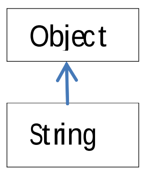
In some cases, class methods are called without dot notation. In fact, the only reason the prefix"Math." is required, is to identify to the compiler where the sqrt() method is defined. (It is defined in the Math class in the java.lang package.) If a method is defined in the same filewhere it is called, then prefixing the method's name with the class name is unnecessary.

Class Hierarchy

Although we'll say a great deal about class hierarchies later, this important topic deserves a quicktour now to alert you to an essential characteristic of Java and other object-oriented programming languages. We noted earlier that the relationship between a class and its package is mostly oneof convenience. (A package is a convenient place to put common classes.) Of far more importance is the relationship of a class to other classes. All classes in Java exist in a hierarchy.Those"further down" the hierarchy inherit characteristics from those "further up" the hierarchy.

At the top of the hierarchy is the Object class. All Java classes inherit characteristics from the Object class. An example of a class "just below" the Object class is the String class. In this relationship, the Object class in the superclass, and the String class is a subclass. Figure a illustrates this. By convention, the arrow points to the superclass. It is common to show classhierarchies horizontally, as in Figure b, since more classes can be placed in a single illustration.

(a)



(b)

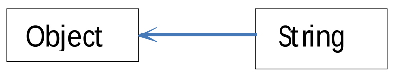


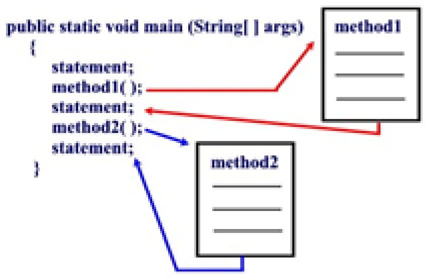
Figure : Class hierarchy (a) vertical illustration (b) horizontal illustration

3.6 Predefined Java methods, Programmers Defined Methods

A method is a set of code which is referred to by name and can be called (invoked) at any point in a program simply by utilizing the method's name. Think of a method as a subprogram that acts on data and often returns a value.

Each method has its own name. When that name is encountered in a program, the execution of the program branches to the body of that method. When the method is finished, execution returns to the area of the program code from which it was called, and the program continues on

To the nest line of code



Good programmers write in a modular fashion which allows for several programmers to work independently on separate concepts which can be assembled at a later date to create the entire project. The use of methods will be our first step in the direction of modular programming.

Methods are time savers, in that they allow for the repetition of sections of code without retyping the code. In addition, methods can be saved and utilized again and again in newly developed programs.

Pre-defined Methods

o Java has a library of (pre-defined) classes and methods, organized in packages (e.g. java.util.Scanner).

o In order to use them, we "import" the packages (or classes individually).

o By default, the java.lang package is automatically imported, in any Java programs.

o In particular, classes String, Math and Character are included in java.lang.

o A method is a sub-routine which performs a certain task and returns the result.

o A method has three parts:

returnTypemethodname(parameters);

e.g. The method pow (for power) in class Math

doublepow(double x, double y) -- This method computes and returns x to the power of y.

To use this method, you call the method with the class name, followed by .and the method name, and appropriate parameters in parentheses. And you receive the returned value in a variable.

double d1 = 1.5, d2 = 2.0;

double result;

result = Math.pow(d1, d2); // call the method with parameters d1 & d2, received the returned value in result//

Built-in: Build-in methods are part of the compiler package, such as System.out.println( ) and System.exit(0).

User-defined Methods

Basics

• In addition to using java pre-defined methods, users (programmers) can write their own methods.

• To write a method, programmers first decides a sub-routine to make as a method. Usually, a useful task, for example those which are used more than once in a program, are selected to be a method.

- Then programmers write the definition of the method by providing:
    - o the three parts (name of the method, parameters and the return type); and
    - o the body of the method -- the actual routine of the task.

User-defined: User-defined methods are created by you, the programmer. These methods take-on names that you assign to them and perform tasks that you create.

Syntax:

```
public static <return-type><name>(<parameter definitions>)
{
<body>
}
```

User-defined methods in a program are usually written below main. They must be also 'public static' method.

```
  // filename: MyProg.java
importjava.util.Scanner;
public class MyProg
  {
public static void main(String[] args)
    {
      Scanner sc = new Scanner(System.in);
System.out.print("Enter two integers: ");
int n1 = sc.nextInt();
int n2 = sc.nextInt();
int max = largerInt(n1, n2); // calls the user-defined method
System.out.println(max + " is larger of the two.");
    } // end main
    // A user-defined method largerInt.
    // This method takes two parameters, both are of type int.
    // The method returns an int, which is the larger of the two parameters.
public staticintlargerInt(int x, int y)
    {
int larger;
if (x > y)
larger = x;
else
larger = y;
return larger;  // return statement at the end
    }
}
```

For the return type, a method can return a value or return nothing at all (void return type).

Void methods do NOT have any return statement.

```
public class MyProg2
  {
public static void main(String[] args)
    {
     Scanner sc = new Scanner(System.in);
System.out.print("Enter two integers: ");
int n1 = sc.nextInt();
int n2 = sc.nextInt();
printTwoIntegers(n1, n2); // call the method, but not receive returned value
    } // end main
   // A void method printTwoIntegers.
public static voidprintTwoIntegers(int x, int y)
    {
System.out.println(x + " " + y);
     // no return statement in the body
    }
}
```

For parameters, a method may take no parameters. Even then, the parentheses must be given (with nothing inside).

```
public class MyProg3
  {
public static void main(String[] args)
    {
for (inti = 0; i< 3; i++)
bark();
    } // end main
public static void bark()  // no parameters
    {
```

```
System.out.println("woof!");
    }
}
```

3.7 Summary

We have discussed all the basic logical and relational operators available in Java and also seen their use in expressions. Order of precedence of operators during the evaluation of expressions have been highlighted.Finally, chapter deals with predefine and programmers defines Java Methods.

3.8 Self-Assessment Questions

1. Discuss role of Relational Operators in Java.
2. Write Short note on Logical Operator in Java.
3. Define predefine Java Methods with suitable Examples.
4. What do you understand by order of precedence. Illustrate with suitable example, order of precedence in Java?
5. Determine the value of each of the following logical expressions if a=5,b=10 and c=-6
   a) a>b && a<c
   b) a<b && a>c
   c) a==c|| b>a
   d) b>15 && c<0 || a>0

3.9 References

1. Programming with Java A Primer, E.Balaguruswamy Tata McGraw Hill Companies
2. Java Programming John P. Flynt Thomson 2nd
3. Java Programming Language Ken Arnold Pearson
4. The complete reference JAVA2, Herbert schildt. TMH
5. Big Java, Cay Horstmann 2nd edition, Wiley India Edition
6. Core Java, Dietel and Dietel
7. Java - Balaguruswamy
8. Java server programming, Ivan Bayross SPD

Unit – 4
Looping in Java

Structure of the unit

4.0 Objective

This chaper gives the general overview of

- looping in Java
- Controlling of loops in Java
- Basics of classes and methods

4.1 Introduction

A program is a set of statement, which is executed sequentially in the order in which they written .The statements inside your source files are generally executed from top to bottom, in the order that they appear. Control flow statements, however, break up the flow of execution by employing decision making, looping, and branching, enabling your program to conditionally execute particular blocks of code or we can say the process of repeatedly executing a block of statements is known as looping. The looping statements (for, while, do-while), and the branching statements (break, continue, return) are supported by the Java programming language.

4.2 Loops in Java

The programming means generally sequential programming. This means that flow is downward, from top to bottom, with every line of code being executed, unless we tell Java otherwise. Java has very flexible three looping mechanisms. We can use one of the following three loops:

- while Loop
- do...while Loop
- for Loop

4.2.1 While loop

The while loop allows programs to repeat a statement or series of statements, over and over, as long as a certain test condition is true. The syntax for while loop is given below.

```
while(test condition)
{
    block of code;
}
```

The test condition must be enclosed in parentheses. the block of code is known as the body of the loop and is enclosed in braces and indented for readability. (the braces are not required if the body is composed of only one statement.) semi-colons follow the statements within the block only.

when a program encounters a while loop, the test condition is evaluated first. if it is true, the program executes the body of the loop. the program then returns to the test condition and reevaluates. if still true, the body executes again. this cycle of testing and execution continues until the test condition evaluates to false. if you want the loop to eventually terminate, something within the body of the loop must affect the test condition. otherwise, a disastrous infinite loop is the result. the while loop is an entry-condition loop. if the test condition is false to begin with, the program never executes the body of the loop.

example:

```
        Class test {
                public static void main( String args[] ){
                 int a= 10;
                while( a < 15 ){
                System.out.print(“value of a : “ + a );
                a++;
                System.out.print(“\n”);
                        }
                        }
                }
```

This would produce following result:

```
value of a : 10
value of a : 11
value of a : 12
value of a : 13
value of a : 14
```

The following program fragment prints and counts the characters of a string variable:

```
String name=”Abcd “;
int i = 0;                        //begin with the first cell of string
while (i < name.length( ))        // loop test condition
```

```
{
                                        //print each cell on a new line
   System.out.println(name.charAt(i));
        i++;                              //increment counter by 1
}
System.out.println(“There are “+ i + “ characters.”);
```

/*This last statement prints when the test condition is false and the loop terminates*/

4.2.2 Do-while loop:

The do...while Loop is similar to the while loop, except that the test condition occurs at the end of the loop. So that the body of the loop always executes at least once.

```
do
{
   block of code;
}
while (test condition);
```

The test condition must be enclosed in parentheses and followed by a semi-colons . Semi-colons also follow each of the statements within the block. The body of the loop (the block of code) is enclosed in braces and indented for readability. (The braces are not required if only one statement is used in the body of the loop.

So do-while loop is an exit-condition loop. This means that the body of the loop is always executed first. Then, the test condition is evaluated. If the test condition is true, the program executes the body of the loop again. If the test condition is false, the loop terminates and program execution continues with the statement following the while.

/*The following program fragment is an input routine that insists that the user type a correct response — in this case, a small case or capital case ‘Y’ or ‘N’. The do-while loop guarantees that the body of the loop (the question) will always execute at least one time.*/

```
char ans;

do
{
    System.out.println(“Do you want to continue (Y/N)?”);
    System.out.println(“You must type a ‘Y’ or an ‘N’”);
    ans = Console.readChar( );
}
while((ans !=’Y’)&&(ans !=’N’)&&(ans !=’y’)&&(ans !=’n’));
```

/*the loop continues until the user enters the correct response*/

Example:

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

    int x= 15;

    do{

      System.out.print(“value of x : “ + x );

      x++;

      System.out.print(“\n”);
    }while( x < 20 );
}
```

This would produce following result:

```
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

4.2.3 For Loop

The statements in the for loop repeat continuously for a specific number of times. The while and do-while loops repeat until a certain condition is met. The for loop repeats until a specific count is met. Use a for loop when the number of repetitions is known, or can be supplied by the user. The coding format is:

```
for(startExpression; testExpression; countExpression)
{
block of code;
}
```

The startExpression is evaluated before the loop begins. It is acceptable to declare and assign in the startExpression (such as int x = 1;). Java evaluates the startExpression

When the following code is written:

```
for (int x= 1; x<=5; x++)
{
System.out.println(x);
  }
```

x will not be recognized as a variable .The countExpression executes after each trip through the loop. The count may increase/decrease by an increment of 1 or of some other value. The testExpression evaluates to true or false. While TRUE, the body of the loop repeats. When the testExpression becomes false, Java stops looping and the program continues with the statement immediately following the for loop in the body of the program.

Braces are not required if the body of the for loop consists of only one statement. Be aware that when a for loop terminates, the value stored in the computer's memory under the looping variable will be "beyond" the testExpression in the loop. It must be sufficiently large (or small) to cause a false condition.

for(x = 0; x <= 5; x++)

System.out.println("java");

The output of this statement is that it print 6 times java in new line.

When this loop is finished, the value 6 is stored in x.

Common Error:

If you wish the body of a for loop to be executed, DO NOT put a semicolon after the for's parentheses. Doing so will cause the for statement to execute only the counting portion of the loop, creating the illusion of a timing loop. The body of the loop will never be executed.

Semicolons are required inside the parenthesis of the for loop. The for loop is the only statement that requires such semicolon.

4.2.4 Nested loop

The placing of one loop inside the body of another loop is called nesting. When you "nest" two loops, the outer loop takes control of the number of complete repetitions of the inner loop. While all types of loops may be nested, the most commonly nested loops are for loops.

.. A for loop without nesting is shown below:

```
for(num1 = 0; num1 <= 9; num1++)
{
   System.out.println( num1 );
}
```

The two nested loops shown below:

```
for(num2=0;num2<=3;num2++)        //outer loop

{

for(num1 = 0; num1 <= 2; num1++)            // inner loop
{
System.out.println( num1 );
}

}
```

When working with nested loops, the outer loop changes only after the inner loop is completely finished

Let's take a look at a trace of two nested loops. In order to keep the trace manageable, the number of iterations have been shortened.

Memory		Screen	
int num2	int num1	0	0
0	0	0	1
	1	0	2
	2	1	0
	3 end of loop	1	1
1	0	1	2
	1	2	0
	2	2	1
	3 end of loop	3	0
2	0	3	
	1		

```
                                        2
                                        3 end of loop
                            4 end of loop
```

Remember, in the memory, for loops will register a value one beyond (or the step beyond) the requested ending value in order to disengage the loop

An another example for nested for loop :

//The Isosceles Right Triangle (made with capital letters)

```
for (outer = 'F' ; outer >= 'A' ; outer—)
{
    for (inner = 'A' ; inner <= outer; inner++)
    {
        System.out.print(inner);
    }
    System.out.println( );
}
```

output of this example is as fallows :

```
ABCDEF
ABCDE
ABCD
ABC
AB
A
```

4.3 Break

The *break* keyword is used to stop the entire loop. The break keyword must be used inside any loop or a switch statement. The break keyword will stop the execution of the innermost loop and start executing the next line of code after the block.

Or we can say that a break statement gets you out of a loop. The program continues with the next statement immediately following the loop. Break stops only the loop in which it resides. It does not break out of a "nested loop" (a loop within a loop).

The syntax of a break is a single statement inside any loop:

break;

Example:

```
public static void main(String args[]){
     int [] numbers = {10, 20, 30, 40, 50 }
     for(int x : numbers )
       {
       if( x = = 30 )
             break;
      System.out.println( x );
}}
```

This would produce following result:

```
10

20
```

//using

break within a loop

```
// eliminating all the letter x and after x
String A = “breakexample”;
int count = -1;
while (count < 7)
{
       count++;
       if (A.charAt(count)==’x’)
              break;
    System.out.println(A.charAt(count));
}
System.out.println(“This is an example of break
                                          “);
```

Screen display:

```
b

r

e

a

k

e

x


This is an example of break;
```

4.4 Continue

The *continue* keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop. Continue can be use in any looping statement. In a for loop, the continue keyword causes flow of control to immediately jump to the update statement. In a while loop or do/while loop, flow of control immediately jumps to the Boolean expression. Continue;

- continue performs a “jump” to the next test condition in a loop. The test condition is then evaluated as usual, and the loop is executed as long as the test condition remains true. The continue statement may be used ONLY in iteration statements (loops). It serves to bypass a portion of the body of the loop within iteration.

The syntax of a continue is a single statement inside any loop:

```
continue;
```

Example:

```
public class Test {
  public static void main(String args[]){
     int [] numbers = {10, 20, 30, 40, 50};
     for(int x : numbers ){
        if( x == 30 ){
             continue;
        }
       System.out.print( x );
       System.out.print(“\n”);
     }
   }
}
```

This would produce following result:

```
10
20
40
50
```

So break and continue can be used to stop/jump iteration blocks

```
for (int j = 0; j < 100; j++)
{
for (k = 0; k < 100; k++) {
if ((j % k)==0) continue OUT;
System.out.println(j);
}
}
```

4.5 Classes and Methods

Classes are user defined *data type* and behave like built-in types. A class is blue print for group of objects that have the same properties and common behavior. A class can have many objects. When we create class in java the first step is keyword class and then name of the class or identifier we can say. Next is class body which starts with curly braces {} and between this all things related with that class means their property and method will come here.

```
class <classname>
 { <variable declaration;>
       <method declaration;>
```

}

Members of Class
When we create a class its totally incomplete without defining any member of this class same like we can understand one family is incomplete if they have no members.

Field
Field is nothing but the property of the class or object which we are going to create .for example if we are creating a class called computer then they have property like mem_size, hd_size, os_type etc

Object

Objects are the basic run time entity or in other words object is a instance of a class . An object is a software bundle of variables and related methods of the special class.

Syntax for the Object :

class_name object_name = new class_name();

Ex :Let Fruit is a class and Mango is object. Fruit mango; //declare mango =**new** Fruit();// instantiate

Method
Method is nothing but the operation that an object can perform it define the behavior of object how an object can interact with outside world .startMethod (), shutdownMethod (). Access Level of members: Access level is nothing but where we can use that members of the class.
Each field and method has an access level:

- private: accessible only in this class
- protected: accessible only in this package and in all subclasses of this class
- public: accessible everywhere this class is available
- package or default: accessible only in this package

The general form of a class definition is shown here:

Example :

```
    class classname {
     type instance-variable1;
            type instance-variable2;
            // ...
           type instance-variableN;
           type methodname1(parameter-list) {
                      // body of method
                      }
           type methodname2(parameter-list) {
                      // body of method
                      }
                  // ...
```

```
      type methodnameN(parameter-list) {
                // body of method
                }
        }
```

The data, or variables, defined within a class are called instance variables. The code is contained within methods. Collectively, the methods and variables defined within a class are called members of the class. In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Thus, it is the methods that determine how a class' data can be used.

As we know now, each object has its own copies of the instance variables. This means that if you have two Box objects, each has its own copy of depth, width, and height. It is important to understand that changes to the instance variables of one object have no effect on the instance variables of another. Example :

```
       /* A program that uses the Box class.
         Call this file BoxDemo.java
        */
  class Box {
      double width;
      double height;
      double depth;
   }
       // This class declares an object of type Box.
         class BoxDemo {
     public static void main(String args[]) {
              Box mybox = new Box();
              double vol;
    // assign values to mybox's instance variables
              mybox.width = 10;
              mybox.height = 20;
              mybox.depth = 15;
     // compute volume of box
             vol = mybox.width * mybox.height * mybox.depth;
             System.out.println("Volume is " + vol);
         }
         }
```

You should call the file that contains this program BoxDemo.java, because the main( ) method is in the class called BoxDemo, not the class called Box. When you compile this program, you will find that two .

class files have been created, one for Box and one for BoxDemo. The Java compiler automatically puts each class into its own class file. It is not necessary for both the Box and the BoxDemo class to actually be in the same source file. You could put each class in its own file, called Box.java and BoxDemo.java, respectively. To run this program, you must execute BoxDemo.class. When you do, you will see the following output:

```
Volume is 3000.0
```

An another example of Class and Object initialization showing the Object Oriented concepts in Java :

```
class Cube {
        int length = 10;
        int breadth = 10;
        int height = 10;
        public static int numOfCubes = 0; // static variable
        public static int getNoOfCubes() { //static method
                return numOfCubes;
        }
        public Cube() {
                numOfCubes++; //
        }
}
public class CubeStaticTest {
        public static void main(String args[]) {
                System.out.println("Number of Cube objects = " + Cube.numOfCubes);
                System.out.println("Number of Cube objects = "
                                + Cube.getNoOfCubes());
        }
}
```

Output

```
Number of Cube objects = 0
Number of Cube objects = 0
```

Constructors :It enable the object to initialize itself when it is created.Constructor has same name as class itself.They do not have return type,not even void. Because they return the instance of the class itself. The example of constructor explain below

The complete listing of class declaration:

```
class simple {
// Constructor
        simple(){
         p = 1;
         q = 2;
         r = 3;
        }
        int p,q,r;
        public int addNumbers(int var1, int var2, int var3)
        {
                return var1 + var2 + var3;
        }
        public void displayMessage()
        {
                System.out.println("Display Message");
        }
}
class example1{
        public static void main(String args[])
        {
                // To create a new instance class
                Simple sim = new Simple();
                // To show the result of the addNumbers
                System.out.println("The result is " + Integer.toString(addNumbers(5,1,2)));
                // To display message
                sim.displayMessage();
        }
```

Method Overloading

Methods that have same name, but different parameter lists and different definition. It is used when object has to perform similar task but using different input parameter. Method overloading results when two or more methods in the same class have the same name but different parameters. Methods with the same name must differ in their types or number of parameters. This allows the compiler to match parameters and choose the correct method when a number of choices exist. Changing just the return type is not enough to overload a method, and will be a compile-time error. They must have a different signature. When no method matching the input parameters is found, the compiler attempts to convert the input parameters to types of greater precision. A match may then be found without error. At compile time, the right implementation is chosen based on the signature of the method call

```
class Overload {
void test(int a) {
System.out.println("a: " + a);
}
void test(int a, int b) {
System.out.println("a and b: " + a + "," + b);
}
double test(double a) {
```

```
System.out.println("double a: " + a);
return a*a;
}
}
class MethodOverloading {
public static void main(String args[]) {
Overload overload = new Overload();
double result;
overload.test(10);
overload.test(10, 20);
result = overload.test(5.5);
System.out.println("Result : " + result);
}

 }
```

Output will be displayed as:

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

C:\NewExamples>java MethodOverloading

a:  10

a: and b :   10,20

double   a: 5.5

result :  30.25
```

Example to calculate area and parameter of a rectangle

```
class square{
  int sqarea(int side){
  int area = side * side;
  return(area);
  }
  int sqpari(int side){
  int pari = 4 * side;
  return(pari);
  }
}
class rectangle{
  int rectarea(int length,int breadth){
  int area = length * breadth;
  return(area);
  }
  int rectpari(int length,int breadth){
  int pari = 2*(length + breadth);
  return(pari);
  }
}
public class ObjectClass{
 public static void main(String args[]){
  int sarea1,sarea2,pari1,pari2;
 int rectarea1,rectarea2,rectpari1,rectpari2;
  square sq = new square();
  rectangle rect = new rectangle();
  int a=20;
  System.out.println(“Side of first square = “ + a);
  sarea1 = sq.sqarea(a);
  pari1 = sq.sqpari(a);
  System.out.println(“Area of first square = “ + sarea1);
  System.out.println(“Parimeter of first square = “ + pari1);
  a = 30;
    System.out.println(“Side of second square = “ + a);
  sarea2 = sq.sqarea(a);
  pari2 = sq.sqpari(a);
  System.out.println(“Area of second square = “ + sarea2);
  System.out.println(“Parimeter of second square = “ + pari2);
  int x = 10, y = 20;
  System.out.println(“Length of first Rectangle = “ + x);
   System.out.println(“Breadth of first Rectangle = “ + y);
  rectarea1 = rect.rectarea(x,y);
  rectpari1 = rect.rectpari(x,y);
  System.out.println(“Area of first Rectangle = “ + rectarea1);
  System.out.println(“Parimeter of first Rectangle = “ + rectpari1);
  x = 15;
  y = 25;
  System.out.println(“Length of second Rectangle = “ + x);
  System.out.println(“Breadth of second Rectangle = “ + y);
  rectarea2 = rect.rectarea(x,y);
  rectpari2 = rect.rectpari(x,y);
  System.out.println(“Area of second Rectangle = “ + rectarea2);
  System.out.println(“Parimeter of first Rectangle = “ + rectpari2);
  }
}
```

Descriptions of the program

Output of the program

```
C:\javac ObjectClass.java

C:\java ObjectClass
Side of first square = 20
Area of first square = 400
Parameter of first square = 80
Side of second square = 30
Area of second square = 900
Parameter of second square = 120
Length of first Rectangle = 10
Breadth of first Rectangle = 20
Area of first Rectangle = 200
Parameter of first Rectangle = 60
Length of second Rectangle = 15
Breadth of second Rectangle = 25
```

```
Area of second Rectangle = 375
Parameter of first Rectangle =80
```

Overriding Methods

A method define in the subclass, has **same name, same argument** and **same return type** as a method in the supperclass. Then , when that method is called, the method defined in the subclass is invoked and executed instead of the one in the supperclass.

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

Output will be displayed as:

Output will be displayed as:

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

C:\NewExamples>java MethodOverriding

Sum of a  and b is  :  30

Sum of a  ,b and c is  :  60
```

Question 1: Consider the following function. This function was intended to create and return a string that is the reverse of its parameter. (For example, if parameter S = "hello", the function should return "olleh".) However, there are several problems with the function as written.

```
static String Reverse(String S) {

   String newS = "";
   char c = S[0];
   while (c) {
       newS = c + newS;
        c++;
    }
   return newS;
}
```

First, identify each problem (write the bad code and give a brief explanation of why it is incorrect). Then give a new, correct version of function Reverse.

4.6 Summary

Repetition of the same operation is called iteration or looping. A for loop can be used to do the same operation to every element of an array. for loops are ideal to use with arrays, where you exactly the number of iterations. When you want repetition and you don't know in advance how many times the repetition will occur you can use recursion or a while loop construct. It is a matter of taste whether you use while loops or

recursion when you don't know beforehand how many times you need to repeat.

Like recursion generalized loops can go infinite. Why writing code you must ensure that your code will terminate.

To group a few items of (possibly) different types together classis used. Classes are user defined *data type* and behave like built-in types. A class is blue print for group of objects that have the same properties and common behavior. A class can have many objects. When we create class in java the first step is keyword class and then name of the class or identifier we can say.

Method is nothing but the operation that an object can perform it define the behavior of object how an object can interact with outside world .

Methods that have same name, but different parameter lists and different definition. It is used when object has to perform similar task but using different input parameter. Method overloading results when two or more methods in the same class have the same name but different parameters.

A method define in the subclass, has **same name, same argument** and **same return type** as a method in the supperclass is method overriding . Then , when that method is called, the method defined in the subclass is invoked and executed instead of the one in the supperclass.

4.7 Self-Assesment Questions

1. Explain the most basic looping statement supported by the Java programming language.
2. Explain the difference between **do-while** statement and while statement. With suitable example.
3. How do you write an infinite loop using the for statement?
4. How do you write an infinite loop using the while statement?
5. Define class and object. Explain them with an example using java

6. Explain class vs. instance with example using java.
7. Difference between instance variable and a class variable
8. Define an abstract class. Explain its purpose.
9. What is the difference between the String and StringBuffer classes?
10. What is the difference between a static and a non-static inner class?

4.8 References

1. Programming with Java A Primer, E.Balaguruswamy Tata McGraw Hill Companies
2. Java Programming John P. Flynt Thomson 2nd
3. Java Programming Language Ken Arnold Pearson
4. The complete reference JAVA2, Herbert schildt. TMH
5. Big Java, Cay Horstmann 2nd edition, Wiley India Edition
6. Core Java, Dietel and Dietel
7. http:\\docs.oracle.com\javase\tutorial\java\javaoo\classvars.html
8. http:\\www.tutorialspoint.com\java\java_loop_control.html

Unit – 5
Arrays in Java

Structure of the Unit

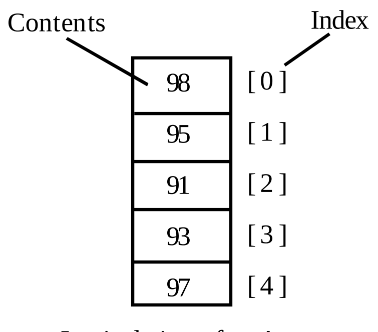
5.0 Objective

The objective of this unit is to

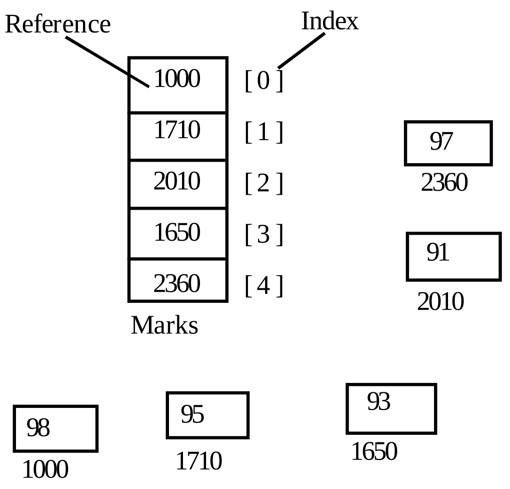
- make student familiar with storing data in a static memory location
- understand one and multidimensional array
- working of character array and string
- formatting data for output

5.1 Introduction

Array is the most important thing in any programming language. An array is a group of variables that share the same name and are ordered sequentially from zero to one less than the number of variables in the array. The number of variables that can be stored in an array is called the array's *dimension*. Each variable in the array is called an *element* of the array. By definition, array is the static memory allocation. It allocates the memory for the same data type in sequence. It contains multiple values of same types. It also store the values in memory at the fixed size. Multiple types of arrays are used in any programming language such as: one - dimensional, two - dimensional or can say multi - dimensional.



Logical view of an Array



Physical view of Array

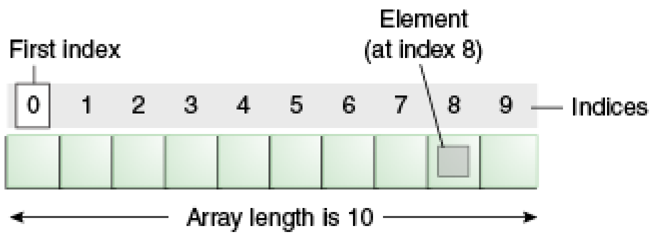
Though you view the array as cells of values, internally they are cells of variables. A java array is a group of variables referenced by a common name. Those variables are just references to a address and will not have a name. These are called the 'components' of a java array. All the components must be of same type and that is called as 'component type' of that java array.

5.2 The One dimensional array

As we know that a n array is a group of variables that share the same name . The number of variables that can be stored in an array is called the array's *dimension*. Using and array in your program is a following steps are required -

5.2.1 Declaration Array

An *array* is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed.



An array of ten elements

Each item in an array is called an *element*, and each element is accessed by its numerical *index*. As shown in the above illustration, numbering begins with 0. The 9th element, for example, would therefore be accessed at index 8.

int [ ]marks;

or

int marks[ ];

5.2.2 Instantiation Array

Inside the square bracket says that you are going to store number of values and is the size of the array 'n'. When you refer the array values, the index starts from 0 'zero' to 'n-12. An array index is always a whole number and it can be a int, short, byte, or char.

Once an array is instantiated, it size cannot be changed. Using the length field like can access its size.

We can declare and allocate an array at the same time like this:

int[] k = new int[3];
float[] yt = new float[7];
String[] names = new String[50];

We can even declare, allocate, and initialize an array at the same time providing a list of the initial values inside brackets like so:

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

float[] yt = {0.0f, 1.2f, 3.4f, -9.87f, 65.4f, 0.0f, 567.9f};

Java array initialization and instantiation together int marks[] = {98, 95, 91, 93, 97}:

An example to sort a Java array—

java api Arrays contains static methods for sorting. It is a best practice to use them always to sort an array.

```
import java.util.Arrays;
public class ArraySort {
 public static void main(String args[]) {
   int marks[] = { 98, 95, 91, 93, 97 };
  System.out.println("Before sorting: " + Arrays.toString(marks));
  Arrays.sort(marks);
  System.out.println("After sorting: " + Arrays.toString(marks));
 }
}
```

Output look like

```
//Before sorting: [98, 95, 91, 93, 97]
//After sorting: [91, 93, 95, 97, 98]
```

The following program, ArrayDemo, creates an array of integers, puts some values in it, and prints each value to standard output.

```
class ArrayDemo {
  public static void main(String[] args) {
       int[] anArray;                // declares an array of integers
     anArray = new int[10];  // allocates memory for 5 integers
       anArray[0] = 100;          //initialize first element
     anArray[1] = 200;     // initialize second element
      anArray[2] = 300;           // etc.
      anArray[3] = 400;
     anArray[4] = 500;
         System.out.println("Element at index 0:  "                 + anArray[0]);
     System.out.println("Element at index 1:  "                 + anArray[1]);
     System.out.println("Element at index 2: "                  + anArray[2]);
     System.out.println("Element at index 3: "                  + anArray[3]);
     System.out.println("Element at index 4: "                   + anArray[4]);
      }
}
```

The output from this program is:

```
Element at index 0: 100
Element at index 1: 200
Element at index 2: 300
Element at index 3: 400
```

Element at index 4: 500

Java Array Default Values

After you instantiate an array, default values are automatically assigned to it in the following manner.

byte – default value is zero

short – default value is zero

int – default value is zero

long – default value is zero, 0L.

float – default value is zero, 0.0f.

double – default value is zero, 0.0d.

char – default value is null, '\u00002.

boolean – default value is false.

reference types – default value is null.

Notice in the above list, the primitives and reference types are treated differently. One popular cause of NullPointerException is accessing a null from a java array.

Iterating a Java Array

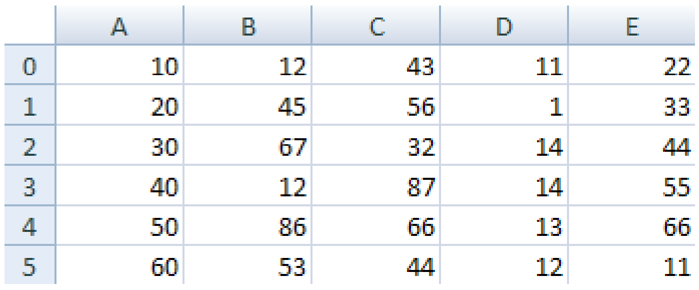
```
public static void main(String args[]) {
    int marks[] = {98, 95, 91, 93, 97};
   //java array iteration using enhanced for loop
   for (int value : marks){
     System.out.println(value);
    }
  }
```

In language C, array of characters is a String but this is not the case in java arrays. But the same behaviour is implemented as a StringBuffer where in the contents are mutable.

5.3 Multidimensional Arrays

The arrays you have been using so far have only held one column of data. But you can set up an array to hold more than one column. These are called multi-dimensional arrays. Or we can say a Multidimensional arrays, are arrays of arrays.

As an example, think of a spreadsheet with rows and columns. If you have 6 rows and 5 columns then your spreadsheet can hold 30 numbers. It might look like this:



	A	B	C	D	E
0	10	12	43	11	22
1	20	45	56	1	33
2	30	67	32	14	44
3	40	12	87	14	55
4	50	86	66	13	66
5	60	53	44	12	11

A multi dimensional array is one that can hold all the values above. You set them up like this:

int[ ][ ] aryNumbers = new int[6][5];

They are set up in the same way as a normal array, except you have two sets of square brackets. The first set of square brackets is for the rows and the second set of square brackets is for the columns. In the above line of code, we're telling Java to set up an array with 6 rows and 5 columns. To hold values in a multi-dimensional array you have to take care to track the rows and columns. Here's some code to fill the first rows of numbers from our spreadsheet image:

```
aryNumbers[0][0] = 10;
aryNumbers[0][1] = 12;
aryNumbers[0][2] = 43;
aryNumbers[0][3] = 11;
aryNumbers[0][4] = 22;
```

So the first row is row 0. The columns then go from 0 to 4, which is 5 items. To fill the second row, it would be this:

```
aryNumbers[1][0] = 20;
aryNumbers[1][1] = 45;
aryNumbers[1][2] = 56;
aryNumbers[1][3] = 1;
aryNumbers[1][4] = 33;
```

The column numbers are the same, but the row numbers are now all 1.

To access all the items in a multi-dimensional array the technique is to use one loop inside of another.

We can also use 3D array .The syntax for three dimensional arrays is a direct extension of that for two-dimensional arrays. Here's a program that declares, allocates and initializes a three-dimensional array:

```
class Fill3DArray {

 public static void main (String args[]) {
   int[][][] M;
   M = new int[4][5][3];
   for (int row=0; row < 4; row++) {
     for (int col=0; col < 5; col++) {
       for (int ver=0; ver < 3; ver++) {
         M[row][col][ver] = row+col+ver;
       }
     }
   }
  }
}
```

5.4 Working with Characters and Strings

String manipulation is an important part of java programming. String can be represented as a sequence of characters array. The following way we can represent character array.

```
Char chararray[ ] =new char [4];

chararray[0]='J';

chararray[1]='A';
```

```
 chararray[2]='V';
chararray[3]='A';
```

Example :

```
public class chararray {
public static void main(String[] args){
   char[] myName=new char [10];
   myName[0]= 'r';
   myName[1]= 'a';
   myName[2]= 'm;
     System.out.println("my name is:");
   for (int i=0; i< myName.length;i++){
      System.out.println(" "+myName[i]);
        }
}
}
```

example: The following program, ArrayCopyDemo, declares an array of char elements, spelling the word "decaffeinated". It uses arraycopy to copy a subsequence of array components into a second array:

```
class ArrayCopyDemo {
   public static void main(String[] args) {
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
                                          'i', 'n', 'a', 't', 'e', 'd' };
        char[] copyTo = new char[7];
       System.arraycopy(copyFrom, 2, copyTo, 0, 7);
      System.out.println(new String(copyTo));
    }
}
```

The output from this program is:

```
Caffeine
```

Example:

```
/*Convert String to Character Array Example This example shows how to convert a given String object to
an array  of character*/
public class StringToCharacterArrayExample {
  public static void main(String args[]){
   //declare the original String object
```

```
    String strOrig = “Hello World”;
    //declare the char array
    char[] stringArray;
      //convert string into array using toCharArray() method of string class
    stringArray = strOrig.toCharArray();
    //display the array
    for(int index=0; index < stringArray.length; index++)
     System.out.print(stringArray[index]);
     }
 }
 /*Output of the program would be :Hello World*/
```

You can place strings of text into arrays. This is done in the same way as for integers:

```
String[ ] aryString = new String[5] ; or
 String arystring ;
arystring =new String (“string”);
```

Example:

```
public static void main(String args[]){
String[ ] aryString = new String[5] ;
aryString[0] = “This”;
aryString[1] = “is”;
aryString[2] = “a”;
aryString[3] = “string”;
aryString[4] = “array”;
for ( int i=0; i < aryString.length; i++ ) {
System.out.println( aryString[i] );
}}
```

The loop goes round and round while the value in the variable called i is less than the length of the array called aryString.

When the above programme is run, the Output window will look like this:

```
This
 is
 a
 string
array
```

5.4.1 Important String Methods

Following are most commonly used methods in the String class.

(a) concat(): This method returns a string with the value of string passed in to the method appended to the end of String which used to invoke the method.

example:

```
public String concat(String s)

String s=”java”;
```

System.out.println(s.concat(“programming”));

Note:-Always use assignment operator in case of concat operator otherwise concat will be unreferenced and you will get old String.example

s.concat(“programming”);

System.out.println(s);

It will present output as “ java” different than what we have expected.So always be careful in using the assignment operator in String method calls.

(b)charAt(): This method returns a specific character located at the String’s specific

index.Remember,String indexes are zero based

Example:

public charAt(int index)

String s=”java programming”;

System.out.println(s.charAt(0));
The output is ‘j’

(c)length():This method returns the length of the String used to invoke the method.

example:

public int length()

String s=”name”;

System.out.println(s.length());

The output is 4

(d) replace(): This method return a String whose value is that of the String to invoke the method ,updated so that any occurrence of the char in the first argument is replaced by the char in the second argument .

public String replace(char old,char new)

Example:

String s=”abc”;

System.out.println(s.replace(‘a’,’A’));

The output is Abc

(e) equalsIgnoreCase(): This method returns a Boolean value depending on whether the value of the string in the argument is the same as the String used to invoke the method.This method will return true even when character in the string object being compared have different cases.

public boolean equalsIgnoreCase(String s)
Example:

String s=”java”;

System.out.println(s.equalsIgnoreCase(“JAVA”));

The output is true

(f) substring(): substring method is used to return a part or substring of the String used to invoke the method.The first argument represents the starting location of the substring.Remember the indexes are zero based.

public String substring(int begin) / public String substring(int begin,int end)

example:

String s="abcdefghi";

System.out.println(s.substring(5));

System.out.println(s.substring(5,8));

The output would be

" fghi "

" fg "

(g) toLowerCase( ): This method returns a string whose value is the String used to invoke the method, but with any uppercase converted to lowercase.:-
public String toLowerCase()
String s="AbcdefghiJ";
System.out.println(s.toLowerCase());
Output is " abcdefghij "
(h) trim( ): This method returns a String whose value is the String used to invoke the method ,but with any leading or trailing blank spaces removed.

public String trim()
Example:
String s="hey here is the blank space ";
System.out.println(s.trim())
The output is " heyhereistheblankspace"

(i) toUpper() : This method returns a String whose value is String used to invoke the method, but with any lowercase character converted to uppercase.

public String toUpperCase()

Example:

String s="AAAAbbbbb";

System.out.println(s.to UpperCase());

The output is " aaaabbbbb "

Above mentioned String methods are most commonly used in String class.Do remember them or otherwise I am always up for you and of course you can refer this site every time you need something.

5.5 Formatted Data Output

Java 5 implements formatted output with printf() .Earlier you saw the use of the print and println methods for printing strings to standard output (System.out). Since all numbers can be converted to strings. format(). Amazingly, there was no built-in way to right justify numbers in Java until Java 5. You had to use if or while to build the padding yoursefl. Java 5 now provides the format() method (and in some cases also the equivalent printf() method from C).. The format() method's first parameter is a string that specifies how to convert a number. For integers you would typically use a "%" followed by the number of columns you want the integer to be right justified in, followed by a decimal conversion specifier “d”. The second parameter would be the number you want to convert.

For example,

```
int n = 2;
System.out.format(“%3d”, n);
```

This would print the number in three columns that is with two blanks followed by 2.

You can put other non-% characters in front or back of the conversion specification, and they will simply appear literally. For example,

```
int n = 2;
System.out.format(“| %3d |”, n);
```

would print

```
|   2 |
```

The following better show you in different way to use format-

```
import java.util.Calendar;
public class TestFormat {
   public static void main(String[] args) {
     long n = 461012;
     System.out.format(“%d%n”, n);      // —>  “461012”
     System.out.format(“%08d%n”, n);   // —>  “00461012”
     System.out.format(“%+8d%n”, n);   // —>  “ +461012”
     System.out.format(“%,8d%n”, n);   // —>  “ 461,012”
     System.out.format(“%+,8d%n%n”, n); // —>  “+461,012”
     double pi = Math.PI;
     System.out.format(“%f%n”, pi);      // —>  “3.141593”
     System.out.format(“%.3f%n”, pi);    // —>  “3.142”
     System.out.format(“%10.3f%n”, pi);  // —>  “     3.142”
     System.out.format(“%-10.3f%n”, pi); // —>  “3.142”
     Calendar c = Calendar.getInstance();
     System.out.format(“%tB %te, %tY%n”, c, c, c); // —>  “May 29, 2006”
     System.out.format(“%tl:%tM %tp%n”, c, c, c);  // —>  “2:34 am”
     System.out.format(“%tD%n”, c);   // —>  “05/29/06”
   }
}
```

Note :

java.util.Calendar//Java Notesjava.util.Calendar The java.util.Calendar class is used to represent the date and time. The year, month, day, hour, minute, second, and milliseconds can all be set or obtained from a Calendar object.

5.6 Summary

An array is the allocation a very common type of data structure where in all elements must be of the same data type. Once defined , the size of an array is fixed and cannot increase to accommodate more elements. The first element of an array starts with zero

Arrays are data structures suitable for problems dealing with large quantities of identically typed data where similar operations need to be performed on every element.

Elements of an array are accessible through their index values. Arrays using a single index are called vectors, those using indices are dimensional arrays. A two dimensional array is really an array of arrays, a 3-dim., an array of arrays of arrays, etc. Arrays have a type associated with them: the type of the elements. The index is always a nonnegativeinteger. Space has to be allocated explicitly for arrays. Either they are initialized with values and then the right amount of space is allocated or the keyword new is used to specify of space.

5.7 Self-Assesment Questions

1. What is an Array?
2. Set up an array to hold the following values, and in this order: 23, 6, 47, 35, 2, 14.
3. Write a program to get the average of all 6 numbers.
4. Using the above values, have your program print out the highest number in the array.
5. Set up an array to hold the following values, and in this order: 23, 6, 40, 36, 12, 4.
6. Using the same array above, have your program print out only the odd numbers.
7. Write a program to use multi dimension array and display output like _

   Mr. Smith

   Ms. Jones

5.8 References

1. Programming with Java A Primer, E.Balaguruswamy Tata McGraw Hill Companies
2. Java Programming John P. Flynt Thomson 2nd
3. Java Programming Language Ken Arnold Pearson
4. The complete reference JAVA2, Herbert schildt. TMH
5. Big Java, Cay Horstmann 2nd edition, Wiley India Edition
6. Core Java, Dietel and Dietel
7. Java – Balaguruswamy
8. Java server programming, Ivan Bayross SPD

Unit - 6
Object Based Programming

Structure of the Unit

6.0 Objective

This chapter provides a general overview of

- Object Based Programming
- Concept of Class, Constructor
- Basis of Composition and Garbage Collection
- Package Creation & Access Method
- Data Abstraction and Encapsulation

6.1 Introduction

Object-oriented programming involves two common senses. One is "has-a" relationship, the other is "is-a" relationship.

The "has-a" relationship describes the member field of an object. If you can talk something with word "has", such thing can be described as a member field in an object in Java code.

The "is-a" relationship describes inheritance relationship between objects. If you can talk something with word "is". They can be described with keyword extends.

Everything can be described as an object. Every object has characteristics or states, more formally, properties. The relationships mentioned above are the structure of the object.

6.2 Class

A class is the blueprint from which individual objects are created. Classes are the fundamental building blocks of a Java program. Class is a collection of data members and functions. You can define an Employee class having members age and salary as follows.

```
class Employee {
  int age;
  double salary;}
```

Example :

Bicycle class can be defined as

```
class Bicycle {
    int cadence = 0;
    int speed = 0;
    int gear = 1;
    void changeCadence(int newValue) {
         cadence = newValue;
    }
    void changeGear(int newValue) {
         gear = newValue;
    }
    void speedUp(int increment) {
         speed = speed + increment;
    }
    void applyBrakes(int decrement) {
         speed = speed - decrement;
    }
    void printStates() {
        System.out.println("cadence:" +
             cadence + " speed:" +
            speed + " gear:" + gear);
     }
}
```

After defining the class, we can create any instance of class that is known as object. For example after defining Sphere class,

```
class Sphere {
 double radius; // Radius of a sphere
 Sphere() {
  }
 // Class constructor
 Sphere(double theRadius) {
   radius = theRadius; // Set the radius
  }
}
public class MainClass {
 public static void main(String[] arg){
   Sphere sp = new Sphere();
      }
}
```

6.3 Class Scope

The scope of a class member is the part of a class in which it can be seen.

- The scope of a method declared anywhere in a class is the entire class.
- The scope of a variable declared within a class, but outside the methods of the class, is the entire class.
- The scope of a method's formal parameters is the entire method.
- The scope of a variable declared in a block (indicated by braces, { }) extends from the declaration to the closing brace. The braces enclosing a complete class description do not indicate a block; everywhere else, they do.
- The scope of a variable declared in the initialization part of a for loop is the entire for loop.

Class variables and class methods (denoted by the keyword static) can be used anywhere within the class. Instance variables and instance methods can only be used by a object (instance) of the correct type, using the syntax object.variableName or object.methodName(arguments)

6.4 This Keyword and its use

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the this keyword. this can be used inside any method to refer to the current object. That is, this is always a reference to the object on which the method was invoked.

Consider the following version of Box( ) :

// A redundant use of this.

```
Box(double w, double h, double d) {
this.width = w;
this.height = h;
this.depth = d;
}
```

The use of this is redundant, it is used to refer current object. Inside Box( ), this will always refer to the invoking object.

To resolve name space collisions

The keyword this is useful when you need to refer to instance of the class from its method. The keyword helps us to avoid name conflicts. When a local variable has the same name as an instance variable, the local variable hides the instance variable. To avoid the conflict between them this can be used.

```
// Use this to resolve name-space collisions.
Box(double width, double height, double depth) {
this.width = width;
this.height = height;
this.depth = depth;
}
```

In the example, this.width, this.height and this.depth refers to the instance variable width, height and depth while width, height and depth refers to the arguments passed in the method.

Other Uses

this keyword can be used for (It cannot be used with static methods):

1. To get reference of an object through which that method is called within it(instance method).
2. To avoid field shadowed by a method or constructor parameter.
3. To invoke constructor of same class.
4. In case of method overridden, this is used to invoke method of current class.
5. To make reference to an inner class. e.g ClassName.this

6.5 Constructors

Constructor is a method can use to set initial values for field variables. When the object is created, Java calls the constructor first. Any code you have in your constructor will then get executed. You don't need to make any special calls to a constructor method - they happen automatically when you create a new object. Constructor has the same name as the name of the class to which it belongs. Constructor's syntax does not include a return type, since constructors never return a value. Constructors may include parameters of various types.

6.5.1 Purpose and Function

Purpose of constructors is to create an instance of a class. This can also be called creating an object, as in:

```
Box p1 = new Box();
```

The purpose of methods, by contrast, is much more general. A method's basic function is to execute Java code.

6.5.2 Constructor vs. Method

Constructors and methods differ in three aspects of the signature: modifiers, return type, and name.

- **Modifiers:** Like methods, constructors can have any of the access modifiers: public, protected, private, or none (often called package or friendly). Unlike methods, constructors can take only access modifiers. Therefore, constructors cannot be abstract, final, native, static, or synchronized.
- **Return Type:** The return types are very different too. Methods can have any valid return type, or no return type, in which case the return type is given as void. Constructors have no return type, not even void.
- **Name:** Constructors have the same name as their class; by convention, methods use names other than the class name. If the Java program follows normal conventions, methods will start with a lowercase letter, constructors with an uppercase letter. Also, constructor names are usually nouns because class names are usually nouns; method names usually indicate actions.

Examples:

```
// Example of defining a constructor and using it to create a object
class D2 {
  D2 (int n) { System.out.println("Hello");}
}
class D1 {
  public static void main(String[] arg) {
      D2 x = new D2(3);
   }
}
```

6.5.3 Multiple Constructors

There can be more than one constructors in a class. When you have more than one constructors, the constructors are different from each other by their parameters declaration. When the class is initialized, Java will call the right constructor that matches the given arguments and type.

```
// a class with 3 constructors, differing by their parameters
class CT2 {
   CT2 () { System.out.println("empty arg called!");}
   CT2 (int n) { System.out.println("int one called!");}
   CT2 (double n) { System.out.println("double one called!");}
}
class CT1 {
   public static void main(String[] arg) {
// creating 3 objects of CT2.
// when each object are created, Java automatically calls the right constructor by matching the arguments
and type.
        CT2 x1 = new CT2();
        CT2 x2 = new CT2(3);
        CT2 x3 = new CT2(3.0);
    }
}
```

6.5.4 Default Constructors

Every class has a constructor even though none are defined. When a class does not define any constructor, the Java compiler actually automatically creates a do-nothing constructor with 0 parameters. So, if a class B does not define any constructors and when a object of B is created B b = new B(), internally Java calls the default constructor B(), which does nothing.

But if user has already created a constructor with argument, Java compiler did not automatically create the no-parameter and do-nothing constructor.

So, for example, if you have a class B that has a constructor B (int n), you cannot create a object of B by new B() because you didn't define it and Java didn't create it automatically. It is a compilation error.

```
class B {
   int x;
   B (int n) {
       x=n;
     System.out.println("constructor 'B (int n)' called!");
    }
}
public class Cons {
   public static void main(String[] args) {B b = new B();}
}
```

6.6 Constructor Overloading

Constructor overloading in java allows to have more than one constructor inside one Class. Just like in case of method overloading you have multiple methods with same name but different signature, in Constructor overloading you have multiple constructor with different signature with only difference that Constructor doesn't have return type in Java. Those constructor will be called as overloaded constructor. Overloading is also another form of polymorphism in Java which allows to have multiple constructor with different name in one Class in java.

6.6.1 Need

Constructor overloading is useful when we want to construct object via different methods. Arraylist in java is classical example of it. ArrayList has three constructors one is empty, other takes a collection object and one take initial Capacity. These overloaded constructor allows flexibility while create arraylist object. No argument constructor is used when anyone don't know size of arraylist during creation, but if size is known then its best to use overloaded Constructor which takes capacity. Since ArrayList can also be created from another Collection, may be from another List than having another overloaded constructor makes lot of sense. By using overloaded constructor we can convert your ArrayList into Set or any other collection.

Example:

```
class Room
{
        double length,breadth,height;
        Room(double l,double b,double h)
        {
                length=l;
                breadth=b;
                height=h;
        }
        Room()
        {
                length=-1;
                breadth=-1;
                height=-1;
        }
        Room(double len)
        {
                length=breadth=height=len;
        }
        double volume()
        {
                return length*breadth*height;
        }
}
```

Important Points

1. Constructor overloading is similar to method overloading in Java.
2. Overloaded constructor must be called from another constructor only.
3. Remember if an overloaded constructor called, it must be first statement of constructor in java.
4. You can call overloaded constructor by using this() keyword in Java.
5. It is preferable to add no argument default constructor because after adding any constructor compiler will not add default constructor.
6. Its best practice to have one primary constructor and let overloaded constructor calls that.

6.7 Composition

Composition is a way to model objects that contain other objects. It means using instance variables that are references to other objects. With composition, references to the constituent objects become fields of the containing object. Composition takes the relationship one step further by ensuring that the containing object is responsible for the lifetime of the object it holds. If Object X is contained within Object Y, then Object Y is responsible for the creation and destruction of Object X.

For example, class Apple is related to class Fruit because Apple has an instance variable that holds a reference to a Fruit object. In this example, Apple is the front-end class and Fruit is the back-end class. In a composition relationship, the front-end class holds a reference in one of its instance variables to a back-end class.

```
class Fruit {
   //...
}
class Apple {
   private Fruit fruit = new Fruit();
   //...
}
```

6.8 Garbage Collection

In java, new operator is used to allocate memory dynamically to objects. This memory remains till there are references for the use of the object. When an object is no longer referenced by the program, the memory space it occupies can be recycled so that the space is made available for subsequent new objects. This work is done by Garbage Collection. The name "garbage collection" implies that objects no longer needed by the program are "garbage" and can be thrown away. In Garbage Collection technique there is no explicit need to destroy an object as java handles the de-allocation automatically. Means Garbage collection relieves users from the burden of freeing allocated memory.

The task of Garbage Collection is done by Java virtual Machine. This technique has some advantages as well as disadvantages.

Advantages:

- It can increase the productivity, because it saves the user time of chasing down an elusive memory problem. This time can be reused by programmer.
- Garbage collection ensures program integrity. Garbage collection is an important part of Java's security strategy. Java programmers are unable to accidentally (or purposely) crash the Java virtual machine by incorrectly freeing memory.

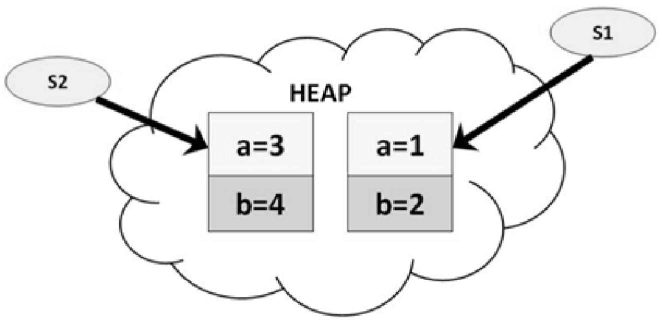
Disadvantages:

- The Java virtual machine has to keep track of which objects are being referenced by the executing program, and finalize and free unreferenced objects on the fly. This activity will likely require more CPU time and it adds an overhead that can affect program performance.
- Programmers in a garbage-collected environment have less control over the scheduling of CPU time devoted to freeing objects that are no longer needed.

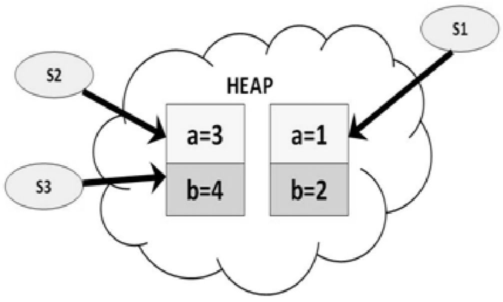
Example:

```
class Student{
int a;
int b;
public void setData(int c,int d){
   a=c;
   b=d;
 }
 public void showData(){
  System.out.println("Value of a = "+a);
  System.out.println("Value of a = "+b);
 }
 public static void main(String args[]){
  Student s1 = new Student();
  Student s2 = new Student();
  s6.setData(1,2);
  s2.setData(3,4);
  s6.showData();
  s2.showData();
  //Student s3;
   //s3=s2;
  //s3.showData();
  //s2=null;
  //s3.showData();
  //s3=null;
  //s3.showData();
 }
}
```

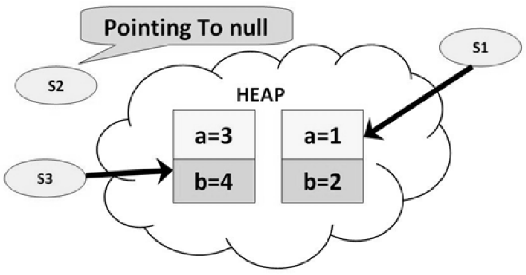
While running the program, As shown in the diagram , two objects and two reference variables are created.



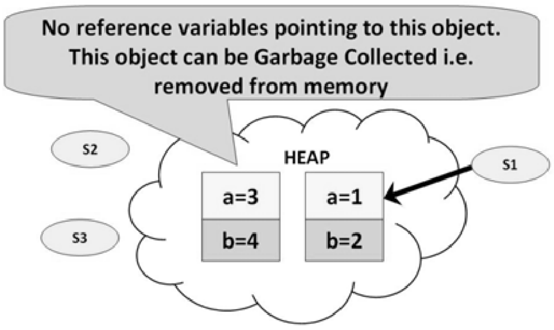
Uncomment line # 20,21,22. Save , compile & run the code. In this case two reference variables are pointing to the same object.



While uncomment line 23 & 24, s2 becomes null , but s3 is still pointing to the object and is not eligible for garbage collection.



After uncomment line 25 & 26, there are no references pointing to the object and becomes eligible for garbage collection. It will be removed from memory and there is no way of retrieving it back.



No reference variables pointing to this object. This object can be Garbage Collected i.e. removed from memory
S2
HEAP
S1
a=3
a=1
S3
b=4
b=2

Important

1) For efficiently using Garbage Collection, assign its reference variable to null.

2) Primitive types are not objects. They cannot be assigned null. So avoid primitive type objects.

6.9 Static Class Members

The static keyword can be used in 3 scenarios

1. static variables
2. static methods
3. static blocks of code.

Static method or a variable is not attached to a particular object, but rather to the class as a whole. They are allocated when the class is loaded. For example in the class StaticVariable each instance has different copy of a class variable. It will be updated each time the instance has been called. Class variable can be called directly inside the main method.

6.9.1 Static Variables

It is a variable which belongs to the class and not to object (instance). Static variables are declared with the static keyword in a class, but outside a method, constructor or a block. Static variables are initialized only once, at the start of the execution and destroyed when the program stops. These variables will be initialized first, before the initialization of any instance variables. There would only be one copy of each class variable per class, regardless of how many objects are created from it.

Default values are same as instance variables. For numbers the default value is 0, for Booleans it is false and for object references it is null. Values can be assigned during the declaration or within the constructor. Additionally values can be assigned in special static initializer blocks.

A static variable can be accessed directly by the class name and doesn't need any object

- Syntax : <class-name>.<variable-name>

Example:

```
import java.io.*;
class Emp{
  // s  variable is a private static variable
  private static double s;
  // DEPT is a constant
  public static final String DEPT = "Development";
  public static void main(String args[]){
      s = 1000;
    System.out.println(DEPT+"average salary:"+s);
   }
}
```

Output :

```
Development average salary:1000
```

6.9.2 Static Method

Similar to static variables, static method belongs to the class and not to the object(instance). A static method can access only static variables and can call only other static method. . Non static variable do not exits in the static method. Static variables are shared by its class instances. Static variable can be used without using the instances of that class. A static method cannot refer to "this" or "super" keywords in anyway.

Syntax : <class-name>.<method-name>

Example

```
class StaticMethod {
  static int j = 0;
  static void getVariable() {
    j++;
   System.out.println("Value of static variable j is"+j);
  }
}
class TestStatic {
 public static void main(String args[]) {
   StaticMethod.getVariable();
  }
}
```

Output:

```
Value of static variable j is 1
```

6.9.3 Static Block

The static block, is a block of statement inside a Java class that will be executed when a class is first loaded in to the JVM.

```
class Test{
 static {
 //Code goes here
 }
}
```

6.10 Final Variables

Final is a keyword or reserved word in java and can be applied to member variables, methods, class and local variables in Java. Once you make a reference final you are not allowed to change that reference and compiler will verify this and raise compilation error if you try to re-initialized final variables in java.

Final Variable

Any member variable or local variable that is declared inside method or block and modified by final keyword is called final variable. Final variables are often declare with static keyword in java and treated as constant. Here is an example of final variable in Java. By default final variable are read only.

Example:

```
Public static final String st = "Hello";
St = new String("Hello") // Compilation error
```

Final Method

Method define with final keyword is called final method and it can not be overridden in sub-class. Any method can be made final in java if it's complete and its behavior should remain constant in sub-classes. Final methods are faster than non-final methods because they are not required to be resolved during run-time and they are bonded on compile time.

Here is an example of final method in Java:

```
class Person{
 public final String getName(){
     return "Smith";
 }
}
Class AnotherPerson extends Person{
   Public final String getName() {
     Return "John"; // compilation error : overridden method is final
     }
}
```

Final Class in Java

Java class with final modifier is called final class in Java. Final class is complete in nature and can not be sub-classed or inherited. Several classes in Java are final e.g. String, Integer and other wrapper classes. Here is an example of final class in java

```
Final class Person{
}
Class AnotherPerson extends Person { //compilation error: cannot inherit from final class
}
```

Advantage of Final Keyword :

6. Final keyword improves performance. Not just JVM can cache final variable but also application can cache frequently use final variables.

2. Final variables are safe to share in multi-threading environment without additional synchronization overhead.

3. Final keyword allows JVM to optimized method, variable or class.

6.11 Packages

A package is a collection of related classes. Package is a method to avoid namespace conflict. Using it, each programmer defines their own namespace and place their code within that namespace, thus two classes that have the exact same name are now distinguishable since they occur in different name spaces. Package provides programmers with greater control over their source code. It would be difficult to manage a few thousand source files in medium to large scale applications. But use of package makes it much easier to manage the source code simply by grouping the related classes into a single package. For example, all the user interface classes of an application will be grouped into a package.

Advantages:

- It shows that the classes and interfaces in the package are related. Often a group of classes and interfaces are related according to functionality so naturally they should be grouped into a package. There are various inbuilt packages in java that bundle classes by function: fundamental classes are in java.lang, classes for reading and writing (input and output) are in java.io.
- It provides an easy way to find the classes you want if they're in a specific package. Just knowing about the functionality of a class you will naturally be able to find it in the right package. If you are looking for an InputStreamReader you will find it in the input and output package, ie: java.io

- It resolves the namespace conflict. It means the names of your classes and interfaces won't be in conflict with those of other programmers. To implement own String class, we have to place it in a package and it will be distinguishable from the java.lang.String class that comes with the Java API.
- Another benefit of using packages is access protection. Suppose a group of different applications utilize the same set of source code, it would make sense to separate this source code and maintain it as a separate library that each application uses. In such a library, there is a public interface and a private interface. The public interface is those classes and methods that are accessible to the application using the library, while the private interface is not accessible to the application. Using a package makes it easier to define which classes form part of the public interface and which are part of the private interface. This makes it easy to control access to certain code sections.

6.11.1 Creating Packages

Creating a Java Package is relatively simple. There are two fundamental requirements for a package to exist:

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

Syntax:

```
package <packageName>;
```

e.g:- package MyPackage;

This should be the first statement of the source file before any imports and class declarations.

It is important to decide which classes and interfaces should belong to package. Anyone is needed to specify this by adding a package declaration to the source file of each class and interface that is a part of the package.

```
package MyPackage;
class xx
{
}
class yy
{
        public static void main(String args[])
  {
  }
}
```

then a directory called MyPackage should be created under the actual directory and the file should stay in that directory. When this particular file is saved it should be saved as yy.java within the directory MyPackage. So when it is compiled the .class files xx.class and yy.class are automatically going to stay in MyPackage directory.

A Hierarchy of packages may be created. To do so, simply each package name should be separated from the one above it by use of a period.

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

But it should also stay in the corresponding directory.

For eg. a package declared as

package java.awt.image

should stay in

java\awt\image directory.

Now, again if a particular. java file is made to stay in a directory because it is written as a part of a corresponding package, then the file should be executed accordingly. For eg. the execution of yy.java in the earlier example should be done as

java MyPackage.yy

being in the actual directory i.e path just above the package.

An Example:-

```
// A simple package
package  MyPackage;
class Balance
    {
                String name;
                double bal;
                Balance(String n, double b)
             {
                name = n;
                bal = b;
             }
          void show( )
             {
                if (bal < 0)
                        System.out.print("--> ");
                System.out.println(name + ": $ " + bal);
               }
    }
class AccountBalance
    {
        public static void main(String args[])
             {
                Balance current[] = new Balance[3];
                current[0] = new Balance("K. J. Fielding", 123.23);
                current[1] = new Balance("Will Tell", 157.02);
                current[0] = new Balance("Tom Jackson", -12.33);
                for (int i = 0; i < 3; i ++) current[i].show( );
              }
        }
```

This file should be called AccountBalance.java and should be placed in a directory named MyPackage. Then after compilation the AccountBalance.class will stay in the MyPackage directory. However, if this does not happen by default then all the .class files should be placed in the MyPackage directory. Then the file should be executed by the command

```
java MyPackage.AccountBalance
```

6.11.2 Naming Packages

Naming a package is therefore a very important aspect of creating a package. This means that package names must be unique, because packages are used to create namespaces that are used to prevent the problem of name conflicts, namespaces must be unique

It is preferable to use domain name, if you have one, and add the project name to it. This will ensure that you have a unique package name, since it is highly unlikely that your organization is working on two projects that have the exact same name. As an example: com.mycompany.myproject

Some companies have decided to drop the top level domain (com, org, net, ...) from their package names for the sake of brevity, this is still perfectly acceptable: mycompany.mypackage

6.11.3 Import Package

To be able to use classes outside of the package it is needed to import the package of those classes. By default, all Java programs import the java.lang.* package. The syntax for importing packages is as follows:

```
import <nameOfPackage>;
```

Example:

Importing a class

```
import java.util.Date;
```

Importing all classes in the java.util package

```
import java.util.*;
```

Using Classes of other packages via fully qualified path

```
public static void main(String[] args)
{
java.util.Date x = new java.util.Date();
}
```

Example:

```
package p2;
import p1;
class Protection2 extends Protection
    {
        Protection2( )
        {
            System.out.println("derived other package constructor");
            // class or package only
            // System.out.println("n = " + n);
            // class only
```

```
                    // System.out.println("n_pri = " + n_pri);
                    System.out.println("n_pro = " + n_pro);
                    System.out.println("n_pub = " + n_pub);
                }
        }
package MyPackage;
/* Now, the Balance class, its constructor, and its show( ) method are public. This means that theycan be
used by non-subclass code outside their packages. */
public class Balance
      {
          String name;
          double bal;
          public Balance (String n, double b)
           {
                    name = n;
                    bal = b;
              }
          public void show( )
           {
                    if (bal < 0)
                              System.out.print("-->");
                    System.out.println(name + ": $" + bal);
              }
         }
import MyPackage.*;
class TestBalance
        {
           public static void main(String args[])
           {
          /* Because Balance is public, the Balance class may be used and its        constructor may be
called. */
          Balance test = new Balance("J. J. Jaspers", 99.88);
           test.show( );    // show( ) may also be called.
              }
          }
```

6.12 Package Access

Access type of a particular member defines the way of accessing that member within or outside the class. For example, if a particular variable is declared private then that variable could be accessed within that class only. It could not be accessed anywhere anyway outside that class. Packages add extra dimension to access control. Packages act as containers for classes and another subordinate packages. Classes act as containers for data and codes. The class is Java's smallest unit of abstraction.

Java addresses four categories of visibility for class members:

- Subclass in the same package
- Non subclasses in the same packages
- Subclasses in different packages
- Classes that are neither in the same package nor subclass

Access specifiers private, public and protected provide a variety of ways. If a variable is made default(not declared public, private or protected) then the variable can be thought to be acting like a private variable outside the package i.e. outside the package that variable cannot be accessed neither from any subclass of that class nor from any other classes. In order to access that variable from outside the package, the variable has to be declared as public or protected. If it is public, then it can be accessed directly from any subclass of that class outside and through an object of that class from some other classes. However if the variable is declared as protected, then it can be accessed outside the package only from any subclass of that class. Moreover, a class has only two possible access levels:- default and public. When a class is declared as public, it is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package.

	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	Yes	Yes

Example:

```
// file Protection.java
package p1;
public class Protection
    {
        int n = 1;
        private int n_pri = 2;
        protected int n_pro = 3;
        public int n_pub = 4;
        public Protection( )
           {
               System.out.println("base constructor");
```

```
                    System.out.println("n = " + n);
                    System.out.println("n_pri = " + n_pri);
                    System.out.println("n_pro = " + n_pro);
                    System.out.println("n_pub = " + n_pub);
                 }
        }
// file Derived.java
package p1;
class Derived extends Protection
     {
          Derived( )
             {
                    System.out.println("derived constructor");
                    System.out.println("n = " + n);
                    // class only
                    // System.out.println("n_pri = " + n_pri);
                    System.out.println("n_pro = " + n_pro);
                    System.out.println("n_pub = " + n_pub);
                 }
        }
// file SamePackage.java
package p1;
class SamePackage
        {
               SamePackage( )
               {
                    Protection p = new Protection( );
                    System.out.println("same package constructor");
                    System.out.println("n = " + p.n);
                    // class only
                    // System.out.println("n_pri = " + p.n_pri);
                    System.out.println("n_pro = " + p.n_pro);
                    System.out.println("n_pub = " + p.n_pub);
                  }
          }
```

```
// file Protection2.java
package p2;
class Protection2 extends p6.Protection
        {
              Protection2( )
             {
                    System.out.println("derived other package constructor");
                    // class or package only
                    // System.out.println("n = " + n);
                    // class only
                    // System.out.println("n_pri = " + n_pri);
                    System.out.println("n_pro = " + n_pro);
                    System.out.println("n_pub = " + n_pub);
                   }
         }
// file OtherPackage.java
package p2;
class OtherPackage
         {
              OtherPackage( )
             {
                    p6.Protection p = new p6.Protection( );
                    System.out.println("other package constructor");
                    // class or package only
                    // System.out.println("n = " + p.n);
                    // class only
                    // System.out.println("n_pri = " + p.n_pri);
                    // class, subclass or package only
                    // System.out.println("n_pro = " + p.n_pro);
                    System.out.println("n_pub = " + p.n_pub);
                   }
}
```

6.13 Data Abstraction Encapsulation

Encapsulation

Encapsulation is information hiding. Information hiding is a technique of hiding the internal implementation detail of an object from its external views. In this technique the internal structure remains private and only way to access services is messages passed via a clearly defined interface. Encapsulation ensures that the object providing service can prevent other objects from manipulating its data or procedures directly, and it enables the object requesting service to ignore the details of how that service is provided. The most important aspect of Information hiding is it allows for implementation of a feature to change without affecting other parts that are depending on it.

Encapsulation makes it easy to maintain and modify code. The client code is not affected when the internal implementation of the code changes as long as the public method signatures are unchanged. For instance:

```
public class Employee
{
private float salary;
public float getSalary()
{
return salary;
}
public void setSalary(float salary)
{
this.salary = salary;
}
```

Example:

```
/* * Java encapsulation - data hiding  */
class Color {
      private int intensity;
      private String name;
      public int getIntensity() {
          return intensity;
        }
      public void setIntensity(int intensity) {
          this.intensity = intensity;
        }
      public String getName() {
           return name;
        }
      public void setName(String name) {
           this.name = name;
        }
}
public class Encapsulation {
   public static void main(String[] args) {
       Color color = new Color();
      //accessing to the members over the methods
      color.setIntensity(5);
      color.setName("black");
      System.out.println("Color: " + color.getName() + " intensity: " + color.getIntensity());
```

```
        //members are declared as private and you can't access directly to them
        //this wont compile
        //color.intensity = 5;
    }
}
```

Abstraction

In simplest terms abstraction can be defines as look at it as "what" the method or a module does not "how" it does it Abstraction is simplifying complex reality by modelling classes appropriate to the problem, and working at the most appropriate level of inheritance for a given aspect of the problem.

In C++, we have to deal with pointers, references and garbage collection. In essence, we have to work on a low level. If we want to implement stack data structure, we have to write different methods like push, pop etc. In Java, those things are abstracted away. To implement Stack, we have to just create the instance of stack class and use inbuilt methods push and pop. In essence, abstraction means that we are working on a higher level.

Abstraction refers to the ability to make a class abstract in OOP. An abstract class is one that cannot be instantiated. All other functionality of the class still exists, and its fields, methods, and constructors are all accessed in the same manner. An instance of the abstract class can not be created. If a class is abstract and cannot be instantiated, the class does not have much use unless it is subclass. A parent class contains the common functionality of a collection of child classes, but the parent class itself is too abstract to be used on its own.

Encapsulation Vs Data Abstraction

Encapsulation facilitates data abstraction, which is the relationship between a class and its data members. Encapsulation encapsulate the data variables and the methods together inside a template called a class Encapsulated data member may be declared public, private, or protected. For high encapsulation all data members of the class should be declared private. That is, the code outside of the class in which the data members are declared can access them only through method calls, and not directly. This is called data abstraction (or data hiding), because now the data is hidden from the user, and the user can have access to it only through the methods. Encapsulation and data abstraction makes the code more reliable, robust, and reusable. This is so because the data and the operations on it (methods) are encapsulated into one entity (the class), and the data member itself and the access to it are separated from each other (tight encapsulation or data abstraction). For example, in tight encapsulation, where your data members are private, if you change the name of a data variable, the access code will still work as long as you don't change the name of the parameters of the method that is used to access it.

In short encapsulation is information hiding and abstraction means working on a higher level, not worrying about the internal details.

6.14 Summary

- This is always a reference to the object on which the method was invoked.
- Constructor is a method can use to set initial values for field variables.
- Constructor overloading in java allows to have more than one constructor inside one Class.
- Constructor overloading is useful when we want to construct object via different methods.
- Static method or a variable is not attached to a particular object, but rather to the class as a whole.

- Final is a keyword or reserved word in java and can be applied to member variables, methods, class and local variables in Java.
- A package is a collection of related classes. Package is a method to avoid namespace conflict.
- Access specifiers private, public and protected provide a variety of ways of using packages and classes.
- encapsulation is information hiding and abstraction means working on a higher level, not worrying about the internal details.

6.15 Self-Assessment Questions

1. Define class and class scope.
2. What is this and what are different uses of it?
3. What role constructor plays in Java? What is constructor overloading?
4. Write a short note on composition and final variable.
5. Explain the working of garbage collection.
6. Discuss the packages in java and their access controls.
7. Differentiate data encapsulation and abstraction.

6.16 References

1. Programming with Java A Primer, E.Balaguruswamy Tata McGraw Hill Companies
2. Java Programming John P. Flynt Thomson 2nd
3. Java Programming Language Ken Arnold Pearson
4. The complete reference JAVA2, Herbert schildt. TMH
5. Big Java, Cay Horstmann 2nd edition, Wiley India Edition
6. Core Java, Dietel and Dietel
7. Java - Balaguruswamy
8. Java server programming, Ivan Bayross SPD

Unit 7

Inheritance and Polymorhism

Structure of the Unit

7.0 Objective

This chapter gives overview of

- Super class and sub class
- Inheritance and use in java
- Concept of abstract classes and methods
- Final class and method
- Wrapper classes

7.1 Introduction

A class is a collection of object sharing the same structure and behavior. The class determines the structure of objects by specifying variables that are contained in each instance of the class, and it determines behavior by providing the instance methods that reflect the behavior of the objects. Most programming languages support this idea, but object oriented programming is based on new concept - to all classes to express the similarities among objects that share some, but not all, of their structure and behavior. The new concept expresses the similarities using inheritance and poly morphism.

7.2 Super Class and Subclass

7.2.1 Super Class

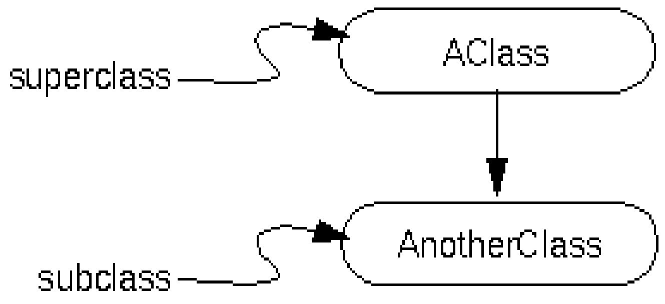
In the relationship between two objects a superclass is the name given to the class that is being inherited from. Super class is also known as parent class. For example "Person" can be a super class. Its state holds the person's name, address, height and weight and has behaviors like habits, hobbies and working style. We could make two new classes that inherit from Person called "Student" and "Worker". They are more specialized versions because although they have names, addresses, habits and hobbies, but they also have characteristics that are different from each other. Worker could have a state that holds a job title and place of employment whereas Student might hold data on an area of study and an institution of learning.

7.2.2 Sub Class

In the relationship between two objects a subclass is the name given to the class that is inheriting from the superclass. It's a more specialized version of the superclass. Subclasses can also be known as derived classes or simply child classes. In the previous example, Student and Worker are the subclasses.

7.2.3 Relation between Super Class and Subclass

In java classes can be derived from other classes. The derived class (the class that is derived from another class) is called a subclass. The class from which its derived is called the superclass.



All classes in java must be derived from some class and they all lead to the top-most class, the class from which all other classes are derived, is the Object class defined in java.lang. Object is the root of a hierarchy of classes.

You can have as many subclasses as you want. There is no limitation to how many subclasses a superclass can have, but in Java, a subclass can only extend one superclass.

7.2.4 Creating Subclass

In class declaration a class is defined as the subclass of another class. For example, suppose that you wanted to create a subclass named SubClass of another class named SuperClass. You would write:

class SubClass extends SuperClass {

```
    .
    .
}
```

This declares that SubClass is the subclass of the Superclass class. It also implicitly declares that SuperClass is the superclass of SubClass. A subclass also inherits variables and methods from its superclass's superclass. The following list itemizes the member variables that are inherited by a subclass:

- Subclasses inherit those member variables declared as public or protected.
- Subclasses inherit those member variables declared with no access specifier as long as the subclass is in the same package as the superclass.
- Subclasses don't inherit a superclass's member variable if the subclass declares a member variable using the same name. The subclass's member variable is said to hide the member variable in the superclass.

Subclasses don't inherit the superclass's private member variables.

7.3 Member Access

- **Public:** Fields, methods and constructors declared public (least restrictive) within a public class are visible to any class in the Java program, whether these classes are in the same package or in another package.
- **Private :** The private (most restrictive) fields or methods cannot be used for classes and Interfaces. It also cannot be used for fields and methods within an interface. Fields, methods or constructors declared private are strictly controlled, which means they cannot be accesses by anywhere outside the enclosing class. A standard design strategy is to make all fields private and provide public getter methods for them.
- **Protected :** The protected fields or methods cannot be used for classes and Interfaces. It also cannot be used for fields and methods within an interface. Fields, methods and constructors declared protected in a superclass can be accessed only by subclasses in other packages. Classes in the same package can also access protected fields, methods and constructors as well, even if they are not a subclass of the protected member's class.
- **Default :** Java provides a default specifier which is used when no access modifier is present. Any class, field, method or constructor that has no declared access modifier is accessible only by classes in the same package. The default modifier is not used for fields and methods within an interface.

7.4 Inheritance

In many cases different types of objects often have some common traits. For example BMW, Innova, Sports car all share the characteristics of car. But each have some additional features that are different from each other like number of seats, number of gates, engine power, etc. Sometimes we need same type of concepts in programming where different objects may have such relations. This can be achieved throw inheritance.

7.4.1 Definition

Inheritance is one of building block of object oriented programming because it allows the creation of hierarchical classifications. Object-oriented programming allows classes to inherit commonly used state and behavior from other classes. In this example, Car now becomes the superclass of BMW, Innova and Sports Car. Even a car class can inherit some properties from a General vehicle class. Here we find that the base class is the vehicle class and the subclass is the more specific car class. Java Inheritance defines an is-a relationship between a superclass and its subclasses. This means that an object of a subclass can be used wherever an object of the superclass can be used. Class Inheritance in java mechanism is used to build new classes from existing classes. The inheritance relationship is transitive: if class x extends class y, then a class z, which extends class x, will also inherit from class y.

The concept of inheritance is used to make the things from general to more specific e.g. When we hear the word vehicle then we got an image in our mind that it moves from one place to another place it is used for traveling or carrying goods but the word vehicle does not specify whether it is two or three or four wheeler because it is a general word. But the word car makes a more specific image in mind than vehicle, that the car has four wheels . It concludes from the example that car is a specific word and vehicle is the general word. If we think technically to this example then vehicle is the super class (or base class or parent class) and car is the subclass or child class because every car has the features of it's parent (in this case vehicle) class.

A subclass must use the extends clause to derive from a super class which must be written in the header of the subclass definition. The subclass inherits members of the superclass and hence promotes code reuse. The subclass itself can add its own new behavior and properties. The java.lang.Objectclass is always at the top of any Class inheritance hierarchy.

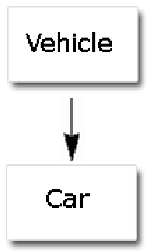
7.4.2 Types of Inheritance

The following kinds of inheritance are there in java.

- Simple Inheritance
- Multilevel Inheritance

Simple Inheritance

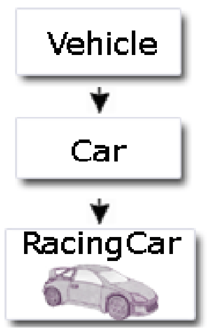
When a subclass is derived simply from it's parent class then this mechanism is known as simple inheritance. In case of simple inheritance there is only a sub class and it's parent class. It is also called single inheritance or one level inheritance.



Multilevel Inheritance

It is the enhancement of the concept of inheritance. When a subclass is derived from a derived class then this mechanism is known as the multilevel inheritance. The derived class is called the subclass or child class for it's parent class and this parent class works as the child class for it's just above (parent) class.

Java does not support multiple inheritance but the multiple inheritance can be achieved by using the interface. In Java Multiple Inheritance can be achieved through use of Interfaces by implementing more than one interfaces in a class.



Syntax

The syntax for creating a subclass is simple. At the beginning of your class declaration, use the extends keyword, followed by the name of the class to inherit from:

```
class SportsCar extends Car {
   // new fields and methods defining
   // a sports car would go here
}
```

This gives SportsCar all the same fields and methods as Car, yet allows its code to focus exclusively on the features that make it unique.

Example 1:

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

Output

```
Volume is : 1000.0
width of MatchBox 1 is 10.0
height of MatchBox 1 is 10.0
depth of MatchBox 1 is 10.0
weight of MatchBox 1 is 10.0
```

Example 2:

```
public class Findareas{
  public static void main (String []agrs){
    Figure f= new Figure(10 , 10);
    Rectangle r= new Rectangle(9 , 5);
    Figure figref;
    figref=f;
    System.out.println("Area is :"+figref.area());
    figref=r;
    System.out.println("Area is :"+figref.area());
  }}
```

```
class Figure{
  double dim1;
  double dim2;
  Figure(double a , double b) {
     dim1=a;
     dim2=b;
   }
  Double area() {
    System.out.println("Inside area for figure.");
    return(dim1*dim2);
   }
}
class Rectangle extends Figure {
  Rectangle(double a, double b) {
     super(a ,b);
   }
  Double area() {
    System.out.println("Inside area for rectangle.");
    return(dim1*dim2);
   }}
```

Output:

```
Inside area for figure.
Area is :100.0
Inside area for rectangle.
Area is :45.0
```

7.4.3 Member Access and Inheritance

Although a subclass includes all the members of its superclass, it cannot access those members of the superclass that have been declared as private. For example, consider the following simple class hierarchy: If you try to compile the following program, you will get the error message.

```
class A {
 private int j; // private to A
}
class B extends A {
 int total;
 void sum() {
   total = j; // ERROR, j is not accessible here
  }
}
```

OutPut : The field A.j is not visible

7.4.4 Super Key Word

Super references the current object just like this, but as an instance of the current class's super class. As the name suggest super is used to access the members of the super class.

It is used for two purposes in java.

- The first use of keyword super is to access the hidden data variables of the super class hidden by the sub class. To access any member use super.member, here member can either be an instance variable or a method.

Example:

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

```
 B subobj = new B(1, 5);
 subobj.Show();
  }
}
```

• The second use of the keyword super in java is to call super class constructor in the subclass. This functionality can be achieved just by using the command super(param-list), here parameter list is the list of the parameter requires by the constructor in the super class.

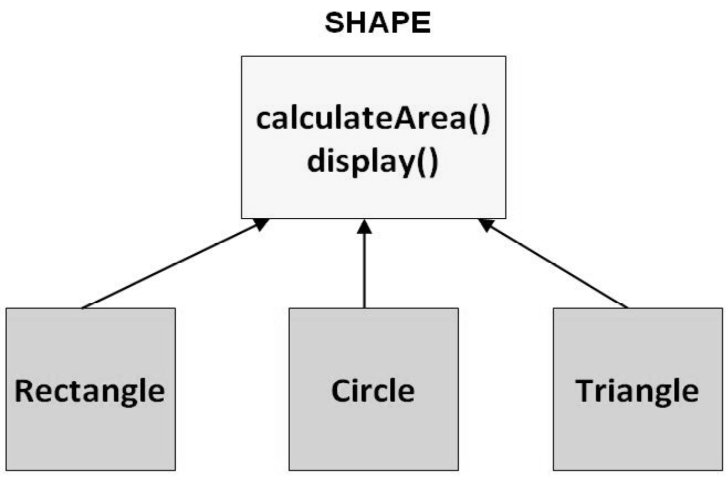
```
class A{
 int a;
 int b;
 int c;
 A(int p, int q, int r){
  a=p;
  b=q;
  c=r;
  }
}
 class B extends A{
 int d;
 B(int l, int m, int n, int o){
 super(l,m,n);
  d=o;
  }
 void Show(){
 System.out.println("a = " + a);
 System.out.println("b = " + b);
 System.out.println("c = " + c);
 System.out.println("d = " + d);
  }
 public static void main(String args[]){
 B b = new B(4,3,8,7);
 b.Show();
  }
  }
```

7.5 Abstract Class and Methods

Class used to declare common characteristics of subclasses is known as Abstract class. An abstract class only be used as a superclass for other classes that extend the abstract class. Abstract classes are used to provide a template or design for concrete subclasses down the inheritance tree.

Abstract classes are useful when we want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.

For example three classes Rect, Circle and Triangle inherit the Shape class. The Shape class is created to save on common attributes and methods shared by the three classes Rectangle , Circle and Triangle. calculateArea() is one such method shared by all 3 child classes and present in Shape class.



Now, assume you write code to create objects for the classes depicted above. Lets observe how these objects will look in a practical world.

An object of the class rectangle, will gives a rectangle, a shape we so commonly observed in everyday life and Circle give a circle, but what about Shape.

Rect obj = new Rect();

Circle obj = new Circle();

Shape obj = new Shape();

If you observe the Shape class serves in our goal of achieving inheritance and polymorphism. But it was not built to be instantiated.Such classes can be labeled Abstract. An abstract class can not be instantiated. Abstract classes are declared with the abstract keyword.

Ex.

```
abstract class Shape{

// code

}
```

Abstract Method

An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void moveTo(double deltaX, double deltaY);
```

If a class includes abstract methods, the class itself must be declared abstract, as in:

```
public abstract class Shape {
  // declare fields
  // declare non-abstract methods
  abstract void draw();
}
```

Example:

First, you declare an abstract class, Shape, to provide member variables and methods that are wholly shared by all subclasses, such as the current position and the moveTo method. Shape also declares abstract methods for methods, such as draw or resize, that need to be implemented by all subclasses but must be implemented in different ways.

```
abstract class Shape {
    int x, y;
    ...
    void moveTo(int newX, int newY) {
        ...
    }
    abstract void draw();
    abstract void resize();
}
```

Each non-abstract subclass of Shape, such as Circle and Rectangle, must provide implementations for the draw and resize methods:

```
class Circle extends Shape {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}

class Rectangle extends Shape {
    void draw() {
        ...
    }
   void resize() {
        ...
    }
Complete Example
abstract class Shape{
abstract void draw();
}
class Rectangle extends Shape{
void draw(){
System.out.println("Drawing Rectangle");
}
}
class Traingle extends Shape{
void draw(){
System.out.println("Drawing Traingle");
}
}
class AbstractDemo{
public static void main(String args[]){
Shape s1=new Rectangle();
s1.draw();
s1=new Traingle();
s1.draw();
}}
```

Result:

```
Drawing Rectangle

Drawing Traingle
```

7.6 Interface

In Java multiple inheritance can be implemented with a powerful construct called interfaces. Interface can be used to define a generic template and then one or more abstract classes to define partial implementations of the interface. Interfaces just specify themethod declaration (implicitly public and abstract) and can only contain fields (which are implicitly public static final). Interface definition begins with a keyword interface. An interface like that of an abstract class cannot be instantiated.

Multiple Inheritance is allowed when extending interfaces i.e. one interface can extend none, one or more interfaces. Java does not support multiple inheritance, but it allows you to extend oneclass and implement many interfaces.

If a class that implements an interface does not define all the methods of the interface, then it must be declared abstract and the method definitions must be provided by the subclass that extends the abstract class.

Example 1: Below is an example of a Shape interface

```
interface Shape {

        public double area();
        public double volume();
}
```

Below is a Point class that implements the Shape interface

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

7.6.1 Abstract Classes versus Interfaces

Unlike interfaces, abstract classes can contain fields that are not static and final, and they can contain implemented methods. Such abstract classes are similar to interfaces, except that they provide a partial implementation, leaving it to subclasses to complete the implementation. If an abstract class contains only abstract method declarations, it should be declared as an interface instead.

Multiple interfaces can be implemented by classes anywhere in the class hierarchy, whether or not they are related to one another in any way. By comparison, abstract classes are most commonly subclassed to share pieces of implementation. A single abstract class is subclassed by similar classes that have a lot in common (the implemented parts of the abstract class), but also have some differences (the abstract methods).

7.7 Final Class and Methods

Final Method

Method define with final keyword is called final method and it can not be overridden in sub-class. Any method can be made final in java if it's complete and its behavior should remain constant in sub-classes. Final methods are faster than non-final methods because they are not required to be resolved during run-time and they are bonded on compile time.

Here is an example of final method in Java:

```
class Person{
 public final String getName(){
      return "Smith";
  }
}
Class AnotherPerson extends Person{
   Public final String getName() {
      Return "John"; // compilation error : overridden method is final
      }
}
```

Final Class in Java

Java class with final modifier is called final class in Java. Final class is complete in nature and can not be sub-classed or inherited. Several classes in Java are final e.g. String, Integer and other wrapper classes. Here is an example of final class in java

```
Final class Person{
}
Class AnotherPerson extends Person { //compilation error: cannot inherit from final class
}
```

7.8 Nested Class

The Java programming language allows you to define a class within another class. Such a class is called a nested class and is illustrated here:

```
class OuterClass {
   ...
   class NestedClass {
       ...
    }
}
```

7.8.1 Advantage

Nested class are used for the following purpose.

Logical grouping of classes-If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.

Increased encapsulation-Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.

More readable, maintainable code-Nesting small classes within top-level classes places the code closer to where it is used.

7.8.2 Types

Nested classes are divided into two categories:

• Static Nested Class: A nested class that is declared static is called a static nested class. Memory to the objects of any static nested classes are allocated independently of any particular outer class object. A static nested class use the instance variables or methods defined in its enclosing class only through an object reference. A static nested class interacts with the instance members of its outer class or any other class just like a top-level class.

```
class OuterClass {
    ...
   static class StaticNestedClass {
        ...
    }
   class InnerClass {
        ...
    }
}
```

• Non Static Nested Class: Non-static nested classes are called inner classes. Inner classes may inherit static members that are not compile-time constants even though they may not declare them.

Example 1:

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

Example 2:

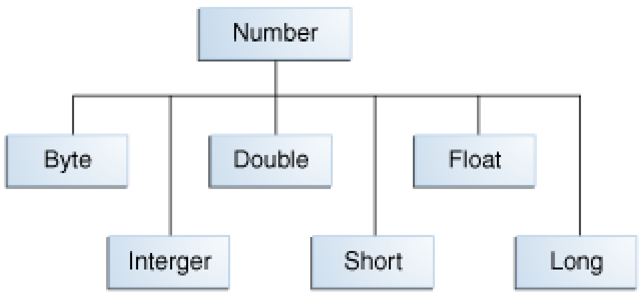
```
//A named inner class is used.
// This is used to show that it can access non-local variables in the enclosing object.
public class InnerExample
```

```
{
static String msg= "Welcome";
public static void main(String[] arg)
{
 outer.new InnerClass().welcome();
}
}
class OuterClass
{
public class InnerClass
{
public void welcome()
{
System.out.println("Welcome from InnerClass()");
}
}
}
```

7.9 Wrappers Classes

Wrapper classes allow primitive data types to be accessed as objects. Wrapper class is a special type of class that's used to make primitive variables into objects, so that the primitives can be included in activities reserved for objects, like as being added to Collections, or returned from a method with an object return value. They are one per primitive type: Boolean, Byte, Character, Double, Float, Integer, Long and Short. There is a wrapper class for every primitive date type in Java. This class encapsulates a single value for the primitive data type. For instance the wrapper class for int is Integer, for float is Float, and so on. Remember that the primitive name is simply the lowercase name of the wrapper except for char, which maps to Character, and int, which maps to Integer.

All of the numeric wrapper classes are subclasses of the abstract class Number:



Purpose:

- To provide a mechanism to "wrap" primitive values in an object so that the primitives can be included in activities reserved for objects, like as being added to Collections, or returned from a method with an object return value.

• To provide an assortment of utility functions for primitives. Most of these functions are related to various conversions: converting primitives to and from String objects, and converting primitives and String objects to and from different bases (or radix), such as binary, octal, and hexadecimal.

The wrapper object of a wrapper class can be created in one of two ways: by instantiating the wrapper class with the new operator or by invoking a static method on the wrapper class.

Example:

All the wrapper classes are declared final. That means you cannot derive a subclass from any of them.All the wrapper classes except Boolean and Character are subclasses of an abstract class called Number, whereas Boolean and Character are derived directly from the Object class.

```
Boolean wboo = new Boolean("false");

Boolean yboo=new Boolean(false);

Byte wbyte = new Byte("2");

Byte ybyte=new Byte(2);

Short wshort = new Short("4");

Short yshort = new Short(4);

Integer wint = new Integer("16");

Integer yint = new Integer(16);

Long wlong = new Long("123");

Long ylong = new Long(123);
```

The value may also be passed as a variable, as shown in the following example:

```
boolean boo = false;

Boolean wboo = new Boolean(boo);

byte b = 2;

Byte wbyte = new Byte(b);

short s = 4;

Short wshort = new Short(s);

int i = 16;

Integer wint = new Integer(i);

long l = 123;

Long wlong = new Long(l);

float f = 12.34f;

Float wfloat = new Float(f);

double d = 12.56d;

Double wdouble = new Double(d);
```

LIST OF WRAPPER CLASS

	Primitive type Wrapper class	
1	boolean	Java.lang.Boolean
2	byte	Java.lang.Byte
3	char	Java.lang.Char
4	double	Java.lang.Double
5	float	Java.lang.Float
6	int	Java.lang.Int
7	long	Java.lang.Long
8	short	Java.lang.Short

7.10 Summary

- The derived class (the class that is derived from another class) is called a subclass. The class from which its derived is called the superclass.
- Inheritance is one of building block of object oriented programming because it allows the creation of hierarchical classifications.
- Super references the current object just like this, but as an instance of the current class's super class.
- Class used to declare common characteristics of subclasses is known as Abstract class.
- Interface can be used to define a generic template and then one or more abstract classes to define partial implementations of the interface.
- Nested Class is defining class with in a class.
- Wrapper class is a special type of class that's used to make primitive variables into objects, so that the primitives can be included in activities reserved for objects

7.11 Self-Assessment Questions

1. Define super class and sub class with suitable examples.
2. What do you understand by inheritance? How many types of inheritance exits?
3. Explain abstract class and methods.
4. Compare and contrast abstract class and interface.
5. Write a short note on Wrapper class.
6. What is nested classes?

7.12 References

1. Programming with Java A Primer, E.Balaguruswamy Tata McGraw Hill Companies
2. Java Programming John P. Flynt Thomson 2nd
3. Java Programming Language Ken Arnold Pearson
4. The complete reference JAVA2, Herbert schildt. TMH
5. Big Java, Cay Horstmann 2nd edition, Wiley India Edition
6. Core Java, Dietel and Dietel
7. Java - Balaguruswamy
8. Java server programming, Ivan Bayross SPD

Unit - 8
Exceptions and Assertions

Structure of the Unit

8.0 Objective

This chapter provide a general overview of

- Concepts such as class, modifiers, interface, packages etc
- What is exception and what is exception handling .
- Differences between Error vs. Exception
- How to throw exceptions out of a method .
- How to write a try-catch block to handle exceptions .
- How to declare custom exception classes.
- To apply assertions to help ensure program correctness .

8.1 Introduction

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

Once code is compiled and running, it will have to face the real world of erroneous input, inexistent files, hardware failure… Such problems are commonly known as runtime errors, which are likely to cause the program to abort. It is therefore important to anticipate such problems and handle them correctly, by avoiding loss of data or premature termination, and by notifying the user.

8.2 Java as an OOP language

There are seven qualities to be satisfied for a programming language to be pure Object Oriented. They are:

1. Encapsulation/Data Hiding
2. Inheritance
3. Polymorphism
4. Abstraction
5. All predefined types are objects
6. All operations are performed by sending messages to objects
7. All user defined types are objects.

In OOPs , you have greater advantage achieved through reusability. Through inheritance , you are utilizing the things that are already available. There is no point in writing which is already available. So , inheritance is one of the cornerstones of oops.

Java is definitely object oriented language. Yaah , it is true that it is hybrid of many object oriented language. But, one should not forget that it has taken best from all the object oriented language. Java has also done great job by removing the features that were causing many problems to programmers.

Java is not a pure Object Based Programming Language because it supports Primitive datatype such as int, byte, long... etc, to be used, which are not objects.

8.3 Defining classes

You've seen classes defined in the following way:

```
class MyClass {
   // field, constructor, and
   // method declarations
}
```

This is a class declaration. The class body (the area between the braces) contains all the code that provides for the life cycle of the objects created from the class: constructors for initializing new objects, declarations for the fields that provide the state of the class and its objects, and methods to implement the behavior of the class and its objects. The preceding class declaration is a minimal one. It contains only those components of a class declaration that are required.

8.4 Modifiers

Modifiers are keywords that you add to those definitions to change their meanings. The Java language has a wide variety of modifiers, including the following:

- Java access Modifiers
- Non Access Modifiers

To use a modifier, you include its keyword in the definition of a class, method, or variable.

Access Control Modifiers:

Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are:

1. Visible to the package. the default. No modifiers are needed.
2. Visible to the class only (private).
3. Visible to the world (public).
4. Visible to the package and all subclasses (protected).

Non Access Modifiers:

Java provides a number of non-access modifiers to achieve many other functionality.

- The static modifier for creating class methods and variables
- The final modifier for finalizing the implementations of classes, methods, and variables.
- The abstract modifier for creating abstract classes and methods.
- The synchronized and volatile modifiers, which are used for threads.

8.5 Packages

A Java package is a mechanism for organizing Java classes into namespaces similar to the modules of Modula. Java packages can be stored in compressed files called JAR files, allowing classes to download faster as a group rather than one at a time. Programmers also typically use packages to organize classes belonging to the same category or providing similar functionality. A package provides a unique namespace for the types it contains. Classes in the same package can access each other's package-access members.

8.6 Interface

In the Java programming language, an interface is a reference type, similar to a class, that can contain only constants, method signatures, and nested types. There are no method bodies. Interfaces cannot be instantiated-they can only be implemented by classes or extended by other interfaces.

8.7 Error

An Error is any unexpected result or behavior obtained from a program during execution. These errors should be handled by programmer, to prevent them from reaching user

Following are list of some causes of errors:

Memory errors - for example memory incorrectly allocated, memory leaks, "null pointer"

File system errors - disk is full, disk has been removed

Network errors - network is down, URL does not exist

Calculation errors - divide by 0

8.7.1 Types of Error

Syntax error arise if language rules not followed. Compiler detect theses errors.

Runtime error during execution arise if environment find an operation that is impossible to carried out.

Logic Error arise when the program does perform in the way it was intended to.

System Errors arise due to internal system errors. Such errors rarely occur. If one does there is little you can do, just notifying the user and trying to terminate the program gracefully.

8.7.2 Error Handling

Method 1. Every method returns a value (flag) indicating either success, failure, or some error condition. The calling method checks the return flag and takes appropriate action.

For example C use this method for almost all library functions (i.e. fopen() returns a valid file or else null)

In this method programmer must remember to check the return value and take appropriate action. This requires lengthy code.

Method 2. Create a global error handling routine, and use some form of "jump" instruction to call this routine when an error occurs.

For example C++, Java adapt it.

"jump" instruction (GoTo) are considered "bad programming practice" and are discouraged. Once you jump to the error routine, you cannot return to the point of origin and so most probably exit the program.

8.8 Exception

An exception is a representation of an error condition or a situation that is not the expected result of a method. Exceptions isolate the code that deals with the error condition from regular program logic. These errors can be caught and handled by your program. Exceptions cannot be ignored, they must be caught otherwise the application will terminate. Exceptions are divided into three major groups:

- ESCAPE requires termination of the operation in progress.
- NOTIFY explicitly forbids termination of the operation.
- SIGNAL allows the invoker to either terminate or resume the operation in progress

8.8.1 Terminology

The language of exceptions is both quite complex and quite inconsistent. Some of the words you will encounter are as follows :

Operation	method which can possibly raise an exception.
Invoker	method which calls operations and handles resulting exceptions.
Exception	concise, complete description of an abnormal event.
Raise	rings an exception from the operation to the invoker, called throw in Java.
Handle	invoker's response to the exception, called catch in Java.
Backtrack	ability to unwind the stack frames from where the exception was raised to the first matching handler in the call stack

8.8.2 Types of Exceptions

Hardware	Generated by the CPU in response to a fault (e.g. divide by zero, overflow, segmentation fault, alignment error, etc).
Software	Defined by the developer to represent any other type of failure. These exceptions often carry much semantic information.
Domain Failure	The inputs, or parameters, to the operation are considered invalid or inappropriate for the requested operation.
Range Failure	Operation cannot continue, or output is possibly incorrect.

8.8.3 Use of Exception

- Enables transfer of control from location of the fault, to code which knows how to handle that particular fault.
- Separates the error management from the normal code.
- Simplifies error processing by allowing entire blocks to be combined inside one exception handler

8.8.4 Keywords for Java Exceptions

throws	Describes the exceptions which can be raised by a method.
throw	Raises an exception to the first available handler in the call stack, unwinding the stack along the way.
try	Marks the start of a block associated with a set of exception handlers.
catch	If the block enclosed by the try generates an exception of this type, control moves here.
Finally	Always called after the try block concludes, or after any necessary catch handler is complete.

8.8.5 Exception Handling

Exception handling separates error-handling code from normal programming tasks, thus making programs easier to read and to modify. Exception handling usually requires more time and resources because it requires instantiating a new exception object, rolling back the call stack, and propagating the errors to the calling methods.

Exception handling is accomplished through the "try - catch" mechanism, or by a "throws" clause in the method declaration.

Try-Catch Mechanism

You should use it to deal with unexpected error conditions. Do not use it to deal with simple, expected situations.

- Wherever your code may trigger an exception, the normal code logic is placed inside a block of code starting with the "try" keyword:
- After the try block, the code to handle the exception should it arise is placed in a block of code starting with the "catch" keyword.
- You may also write an optional "finally" block. This block contains code that is ALWAYS executed, either after the "try" block code, or after the "catch" block code.

Method 1.

Exceptions are handled in Java by the try/catch pair.

Try : Identifies an exception situation Actually creates an exception object containing relevant information.

Catch : An exception handler that knows how to recover from an exception (if only to print out a meaningful message before dying) If you don't catch an exception, the program ends abruptly.

Try/catch is essentially a more flexible syntax than the if () and goto statements based on exceptions being objects themselves (ie. Object orientation)

```
try {

… normal program code

}

catch(Exception e) {

… exception handling code

}
```

Every try block must have at least one catch or finally block attached.

"If any of the code inside the try block throws an exception of the class specified in the catch clause, then:

- The program skips the remainder of the code in the try block.
- The program executes the handler code inside the catch clause.
- If none of the code inside the try block throws an exception, then the program skips the catch clause"
- "It should be obvious, but one beauty of the try/catch statement is that any statement in the try block can assume that all previous statements in the block succeeded. …. If an earlier statement fails, execution jumps immediately to the catch clause; later statements are never executed."

```
        try
        {
        statement 1;
        statement 2;
        statement 3;
}
catch { … }
```

o A try block can be followed by several catch blocks, each dealing with a different type of exception.

```
  try   {
                        loadImage("carleton");
                 }
            catch (EOFException e1)
                 {
                        handleEOF(e1);
                 }
            catch (MalformedURLException e2)
                 {
                        handleMalformedURL(e2);
                 }
```

Method 2 : Passing the exception

In any method that might throw an exception, you may declare the method as "throws" that exception, and thus avoid handling the exception yourself

Example

```
public void myMethod throws IOException {
… normal code with some I/O
}
```

An exception occurs in a method. If you want the exception to be processed by its caller, you should create an exception object and throw it. If you can handle the exception in the method where it occurs, there is no need to throw it.

8.8.6 Throwing Exception

8.8.6.1 Throw

Suppose you want to create a Stack class, with its usual methods: push() and pop()

1. An attempt to pop() from an empty stack will (and should) fail!
2. In that case, pop() should therefore throw an exception…When a method throws an exception, it stops its execution, and goes back to the caller, without returning any value.

```
class Stack
{
          public void pop()
          {
            if (isEmpty())
                    // is the stack empty?
                    {
                     throw new        EmptyStackException();
                    }
                    else
     {
                     // else pop the value
     }
          }
}
```

8.8.6.2 Throws

The throws keyword in java programming language is applicable to a method to indicate that the method raises particular type of exception while being processed.

A method that throws an exception advertises it in the header of the method, using the throws declaration:

public void pop() **throws** **EmptyStackException**

this is mandatory for all exceptions except for RuntimExceptions and its subclasses (because usually you can't predict that they will happen, and if you can, then you can also prevent them!).

Without the throws clause in the signature the Java compiler does not know what to do with the exception.

```
import java.io.IOException;
public class Class1{
public method readingFile(String file) throws IOException{
<statements>
if (error){
throw new IOException("error reading file");
  }
   }
}
```

A method can throw and advertise more than one exception:

```
          public Image loadImage(String s)
                    throws EOFException,
                    MalformedURLException
          { … }
```

Difference between Throw and throws

1. throw is used to throw an exception in a program, explicitly. Whereas, throws is included in the method's declaration part, with a list of exceptions that the method can possible throw.

2. The Throw clause can be used in any part of code where you feel a specific exception needs to be thrown to the calling method. The throws clause tells the compiler that this particular exception would be handled by the calling method.

3. Using throws we only provide information related to unhandled exceptions of a method. throws keyword would not perform any kind of actions.

BUT using 'throw' we explicitly raise an exception and throw would not provide any information rather it perform certain action.

8.8.7 Finally Keyword

When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods. For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The finally keyword is designed to address this contingency. The finally is a Java keyword that is used to define a block that is always executed in a try?catch?finally statement.

The finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception. Any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns. This can be useful for cleanup code(recovers from partial execution of a try block),closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. The finally clause is optional. However, each try statement requires at least one catch or a finally clause.

It is always a good practice to use finally clause after the try and catch block to handle an unexpected exception occurred in the try block.

```
try{
        int x=10/2;
   }
finally{
        system.out.println("i am from finally");
       }
```

here compiler will not compliant.why?even though x=10/2,x=10/0.

8.9 Exception Classes Hierarchy

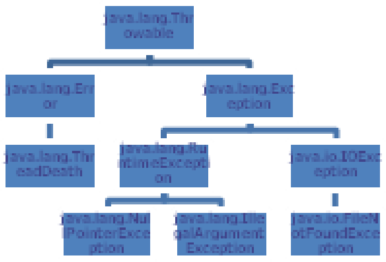
Java exception classes are organised into a hierarchy. There is a basic exception class called Exception as you might expect. But in fact, the base of the hierarchy starts not with Exception but with a class called Throwable, which is then subclassed into Exception and Error.

- Exception subclasses represent errors that a program can reasonably recover from. Except for RuntimeException and its subclasses (see below), they generally represent errors that a program will expect to occur in the normal course of duty: for example, network connection errors and filing system errors.

- Error subclasses represent "serious" errors that a program generally shouldn't expect to catch and recover from. These include conditions such as an expected class file being missing, or an OutOfMemoryError.

• RuntimeException is a further subclass of Exception. RuntimeException and its subclasses are slightly different: they represent exceptions that a program shouldn't generally expect to occur, but could potentially recover from. They represent what are likely to be programming errors rather than errors due to invalid user input or a badly configured environment.

We'll see that in reality, there are exceptions to these uses. For example, there are cases where the organisation of the JDK libraries means that we have to catch an exception that we essentially know will never occur. And there are some cases where we've "got nothing to loose" by trying to catch an Error, even though we may not actually be able to in practice.


java.lang.Throwable
java.lang.Error
java.lang.Exception
java.lang.ThreadDeath
java.lang.RuntimeException
java.io.IOException
java.lang.NullPointerException
java.lang.IllegalArgumentException
java.io.FileNotFoundException

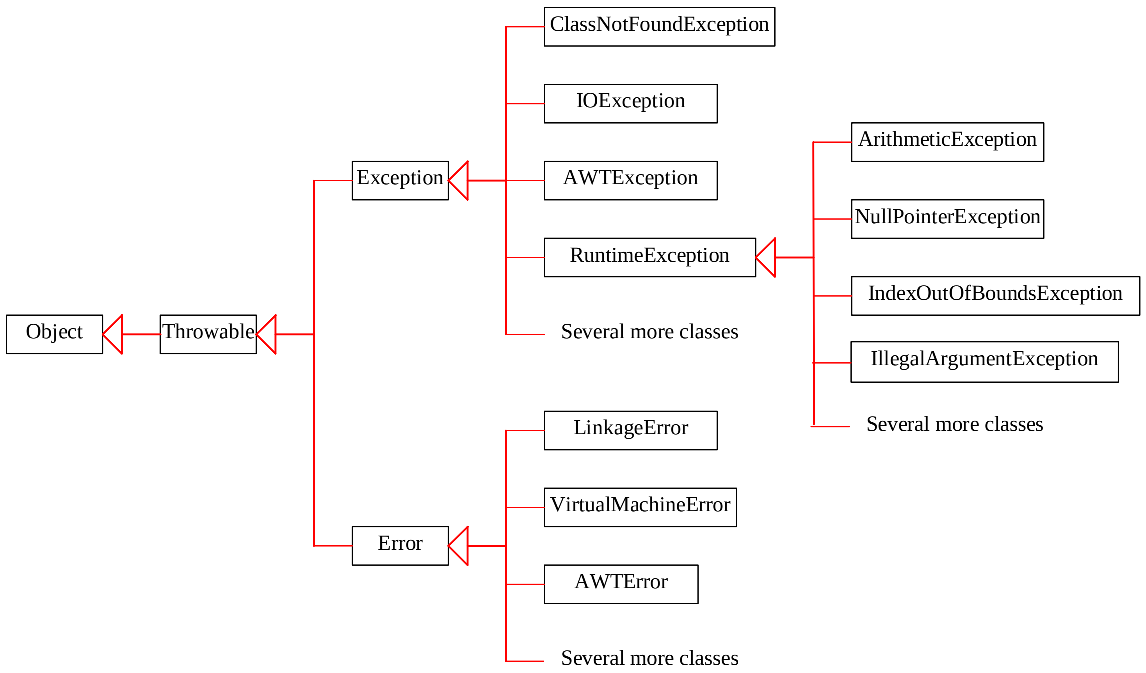
As shown in the UML diagram, exceptions are further divided into 2 groups

runtimeException : Usually (but not only) errors resulting from programming mistakes, Do not need to be advertised or caught. For example

o ArrayIndexOutOfBoundsException,

o NullPointerException

- Exceptions are objects, with state and behavior
- Not just a transient event with no information.
- It has an identity
- Can be referenced, stored, passed about, queried.



8.10 Throwable Class

The java.lang.Throwable is the super class of all errors and exceptions in Java. Only objects of this class can be thrown & caught and handled by try-catch blocks. Similarly, only this class or one of its subclasses can be the argument type in a catch clause.

Instances of two subclasses, Error and Exception, are conventionally used to indicate that exceptional situations have occurred. Typically, these instances are freshly created in the context of the exceptional situation so as to include relevant information

A throwable contains a snapshot of the execution stack of its thread at the time it was created. It can also contain a message string that gives more information about the error. Finally, it can contain a cause: another throwable that caused this throwable to get thrown. The cause facility is new in release 1.4. It is also known as the chained exception facility, as the cause can, itself, have a cause, and so on, leading to a "chain" of exceptions, each caused by another.

A cause can be associated with a throwable in two ways: via a constructor that takes the cause as an argument, or via the initCause(Throwable) method. New throwable classes that wish to allow causes to be associated with them should provide constructors that take a cause and delegate (perhaps indirectly) to one of the Throwable constructors that takes a cause. For example:

```
try {
   lowLevelOp();
}
     catch (LowLevelException le) {
   throw new HighLevelException(le);  // Chaining-aware constructor
}
```

Because the initCause method is public, it allows a cause to be associated with any throwable. For example:

```
 try {
   lowLevelOp();
} catch (LowLevelException le) {
  throw (HighLevelException)
       new HighLevelException().initCause(le);  // Legacy constructor
 }
```

By convention, class Throwable and its subclasses have two constructors, one that takes no arguments and one that takes a String argument that can be used to produce a detail message. Further, those subclasses that might likely have a cause associated with them should have two more constructors, one that takes a Throwable (the cause), and one that takes a String (the detail message) and a Throwable (the cause).

8.10.1 Throwable Constructors

Throwable()	Constructs a new throwable with null as its detail message.
Throwable(String message)	Constructs a new throwable with the specified detail message.
Throwable(String message, Throwable cause)	Constructs a new throwable with the specified detail message and cause
Throwable(Throwable cause)	Constructs a new throwable with the specified cause and a detail message of (cause==null ? null : cause.toString()) (which typically contains the class and detail message of cause).

8.10.2 Throwable methods

Throwable	fillInStackTrace()	Fills in the execution stack trace.
Throwable	getCause()	Returns the cause of this throwable or null if the cause is nonexistent or unknown.
String	getLocalizedMessage()	Creates a localized description of this throwable
String	getMessage()	Returns the detail message string of this throwable.
Throwable	initCause(Throwable cause)	Initializes the cause of this throwable to the specified value.
void	printStackTrace()	Prints this throwable and its backtrace to the standard error stream
void	printStackTrace(PrintStream s)	Prints this throwable and its backtrace to the specified print stream
void	setStackTrace(StackTrace Element[] stackTrace)	Sets the stack trace elements that will be returned by getStackTrace() and printed by printStackTrace() and related methods.
void	printStackTrace(PrintWriter s)	Prints this throwable and its backtrace to the specified print writer.
String	toString()	Returns a short description of this throwable

Advantages of Exceptions

Separating Error-Handling Code from "Regular" Code

Exceptions provide the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program. In traditional programming, error detection, reporting, and handling often lead to confusing practice.

Propagating Errors Up the Call Stack

A second advantage of exceptions is the ability to propagate error reporting up the call stack of methods.

Grouping and Differentiating Error Types

Because all exceptions thrown within a program are objects, the grouping or categorizing of exceptions is a natural outcome of the class hierarchy.

8.11 Assertion

An assertion is a statement in Java that enables to test your assumptions about your program. An assertion contains a Boolean expression that should be true during program execution. Assertions can be used to assure program correctness and avoid logic errors. Assertion should not be used to replace exception handling. Exception handling deals with unusual circumstances during program execution. Assertions are to assure the correctness of the program. Exception handling addresses robustness and assertion addresses correctness. Like exception handling, assertions are not used for normal tests, but for internal consistency and validity checks. Do not use assertions for argument checking in public methods.

The assertion statement has two forms

```
assert Expression1 ;                Where Expression1 is a boolean expression
```

When the system runs the assertion, it evaluates Expression1 and if it is false throws an AssertionError with no details.

The second form of the assertion statement is:

assert Expression1 : Expression2 ;

where:

- Expression1 is a boolean expression
- Expression2 is an expression that has a value

Use the second version of the assert statement to provide a detailed message for the AssertionError. The purpose of the message is to communicate the reason for the assertion failure. The system passes the value of Expression2 to the appropriate AssertionError constructor, which uses the string error message

Since AssertionError is a subclass of Error, when an assertion becomes false, the program displays a message on the console and exits.

Compiling Programs with Assertions

Since assert is a new Java keyword introduced in JDK 1.4, you have to compile the program using a JDK 1.4 compiler. Furthermore, you need to include the switch -source 1.4 in the compiler command as follows:

```
javac -source 1.4 AssertionDemo.java
```

NOTE: If you use JDK 1.5, there is no need to use the -source 1.4 option in the command.

Running Programs with Assertions

By default, the assertions are disabled at runtime. In this case, the assertion has the same semantics as an empty statement

To enable it, use the switch -enableassertions, or -ea for short, as follows:

```
java -ea AssertionDemo
```

Assertions can be selectively enabled or disabled at class level or package level. The disable switch is -disableassertions or -da for short. For example, the following command enables assertions in package package1 and disables assertions in class Class1.

```
java -ea:package1 -da:Class1 AssertionDemo
```

Executing Assertions Example

```
public class AssertionDemo
 {
        public static void main(String[] args)
{
                        int i; int sum = 0;
                              for (i = 0; i < 10; i++)
{
                                    sum += i;
                              }
                             assert i == 10;
                             assert sum > 10 && sum < 5 * 10 : "sum is " + sum;
                      }
}
```

8.12 Summary

Exceptions exist to make your life easier. If you use them, they will.

Exception handling templates are a simple yet powerful mechanism that can increase the quality and readability of your code. It also increases your productivity, since you have much less code to write, and less to worry about. Exceptions are handled by the templates. And, if you need to improve the exception handling later in the development process, you only have a single spot to change it in

Exceptions are not "errors". They are signals that something went wrong. Signals that can be used. Catch them where you can use them the best.

Don't use return values to indicate success or failure. Exceptions won't get ignored as easily as a return value. This will make bugs in the code easier to spot.

Even an uncaught exception that crashes your program can be more use than one that is ignored.

A program can use exceptions to indicate that an error occurred. To throw an exception, use the throw statement and provide it with an exception object - a descendant of Throwable - to provide information about the specific error that occurred. A method that throws an uncaught, checked exception must include a throws clause in its declaration.

A program can catch exceptions by using a combination of the try, catch, and finally blocks.

- The try block identifies a block of code in which an exception can occur.
- The catch block identifies a block of code, known as an exception handler, that can handle a particular type of exception.
- The finally block identifies a block of code that is guaranteed to execute, and is the right place to close files, recover resources, and otherwise clean up after the code enclosed in the try block.

The try statement should contain at least one catch block or a finally block and may have multiple catch blocks.

The class of the exception object indicates the type of exception thrown. The exception object can contain further information about the error, including an error message. With exception chaining, an exception can point to the exception that caused it, which can in turn point to the exception that caused it, and so on.

An assertion is a statement in Java that enables to test your assumptions about your program.

8.13 Self-Assessment Questions

1. What is the difference between Exception and Error in java?
2. What is the difference between throw and throws?
3. What is the importance of finally block in exception handling?
4. Explain the significance of try-catch blocks?
5. What are the different ways to handle exceptions?

8.14 References

1. Programming with Java A Primer, E.Balaguruswamy Tata McGraw Hill Companies
2. Java Programming John P. Flynt Thomson 2nd
3. Java Programming Language Ken Arnold Pearson
4. The complete reference JAVA2, Herbert schildt. TMH
5. Big Java, Cay Horstmann 2nd edition, Wiley India Edition
6. Core Java, Dietel and Dietel
7. Java - Balaguruswamy
8. Java server programming, Ivan Bayross SPD

Unit - 9
Advance concepts in Exception

Structure of the Unit

9.0 Objective

This chapter covers a few techniques foe efficient and less error prone exception handling. In this chapter we have following objective -

- To examine Difference between checked and unchecked exception.
- To write customize exception.
- To rethrow exception.
- To redirect exception
- To find relationship between exception and inheritance

9.1 Introduction

Exceptions are long-established programming structures that allow programmer to handle multiple runtime errors through a centralized mechanism within client code. Certainly, this method for handling errors has proven to be highly successful in the vast majority of applications, which turned it rapidly into the standard technique for delegating error manipulation to specific fragments of code, different from those that generate error conditions. Exception handling helps you to avoid long exception stack traces and to get truly unique error codes for each error in your application.

For advanced exception handling is available to all developers, allowing them to write more robust code by using popular "try-catch" blocks, as well as working with "throw" statements for complex user-defined exceptions.

Before examining exception more closely it should be good to summarize what is exception handling in brief.

9.2 Handling Exception in Java

The term "exception" means "exceptional condition" and is an occurrence that alters the normal program flow. A bunch of things can lead to exceptions, including hardware failures, resource exhaustion, and good old bugs. When an exceptional event occurs in Java, an exception is said to be "thrown." The code that's responsible for doing something about the exception is called an "exception handler," and it "catches and handles" the thrown exception.

Exception handling works by transferring the execution of a program to an appropriate exception handler when an exception occurs.

Exception handling helps us catch or identify abnormal scenarios in our code and handle them appropriately instead of throwing up a random error on the front-end of the application.

Exception handling allows developers to detect errors easily without writing special code to test return values. Even better, it lets us keep exception-handling code cleanly separated from the exception-generating code. It also lets us use the same exception-handling code to deal with a range of possible exceptions.

9.3 Checked and Unchecked Exception

Java exception classes (we will ignore errors here, and focus on exceptions) are categorized as either "checked" or "unchecked". These categorization affect compile-time behavior only; they are handled identically at runtime. We (and Java) can easily determine in which category each exception is defined.

Checked and unchecked exceptions are functionally equivalent. There is nothing you can do with checked exceptions that cannot also be done with unchecked exceptions, and vice versa. Regardless of your choice between checked and unchecked exceptions it is a matter of personal or organisational style. None is functionally better than the other.

There is little reason for the program to try to detect such errors and catch them, or to propagate them out of the method in which they occur to be caught by other methods. Best to have Java automatically catch such exceptions and print an error message including a stack trace, so that the programmer can try to fix the bug.

Use either only checked exceptions or only unchecked exceptions. Mixing exception types often results in confusion and inconsistent use.

One way to distinguish these exceptions is by the "source" of the error: whether it is internal or external to the program's code.

Unchecked exceptions can occur in any statement that accesses a member of an object (NullPointerExceptions).

Difference between checked and unchecked exception

Unchecked Exception	**checked Exception**
Use unchecked exceptions for the errors the application cannot recover from	use checked exceptions for all errors the application can recover from
An unchecked exception is any class that IS A SUBCLASS of RuntimeException	A checked exception is any class that is NOT A SUBCLASS of RuntimeException
Example - IndexOutOfBoundException	Example - EOFException
Unchecked exceptions do not have this requirement.	Checked exceptions must be explicitly caught or propagated.
Unchecked exceptions extend the java.lang.RuntimeException	Checked exceptions in Java extend the java.lang.Exception class.
Unchecked exception are inherited from RuntimeException and no handling is enforced by the compiler. Simply put you don't have to check them.	Checked exceptions must be handled by a try catch mechanism or by adding the throws declaration to the method. If neither of those is done then the compiler throws an exception.
Unchecked exceptions do not have this requirement.	Checked exception show the compiler your awareness of the possibility that your method may throw an exception.

Unchecked Exceptions can occur any time and are actually coding bugs so should actually not be handled using catch block instead should be handled by writing bug-free code.	Checked exception is compile - time check made by the java compiler for surety on java's robustness

Here is a method that throws a checked exception, and another method that calls it:

```
public void storeDataFromUrl(String url){
    try {
        String data = readDataFromUrl(url);
    } catch (BadUrlException e) {
        e.printStackTrace();
    }
}
public String readDataFromUrl(String url)
throws BadUrlException{
    if(isUrlBad(url)){
        throw new BadUrlException("Bad URL: " + url);
    }
    String data = null;
    //read lots of data over HTTP and return
    //it as a String instance.
    return data;
}
```

BadUrlException is a checked exception because it extends java.lang.Exception:

unchecked BadUrlException:

```
public void storeDataFromUrl(String url){
    String data = readDataFromUrl(url);
}
public String readDataFromUrl(String url) {
    if(isUrlBad(url)){
        throw new BadUrlException("Bad URL: " + url);
    }
    String data = null;
    //read lots of data over HTTP and
    //return it as a String instance.
    return data;
}
```

Notice how the readDataFromUrl() method no longer declares that it throws BadUrlException. The storeDataFromUrl() method doesn't have to catch the BadUrlException either. The storeDataFromUrl() method can still choose to catch the exception but it no longer has to, and it no longer has to declare that it propagates the exception.

9.4 Exception and Inheritance

Since Java is an Object Oriented programming language, it allows inheritance of all kinds of classes including Throwable classes.

Using inheritance with exceptions enables an exception handler to catch related errors with concise notation. One approach is to catch each type of pointer or reference to a derived-class exception object individually, but a more concise approach is to catch pointers or references to base-class exception objects instead. Also, catching pointers or references to derived-class exception objects individually is error prone, especially if you forget to test explicitly for one or more of the derived-class pointer or reference types.

Various exception classes can be derived from a common base class, when we created class DivideByZeroException as a derived class of class exception. If a catch handler catches a pointer or reference to an exception object of a base-class type, it also can catch a pointer or reference to all objects of classes publicly derived from that base class-this allows for polymorphic processing of related errors.

Consider this example:

```
class Base
{
   void amethod() { }
}
class Derv extends Base
{
   void amethod() throws Exception { } //compile-time error
   public static void main( String s[] )
   {
      Base b = new Derv();   // line 12
      b.amethod();           // line 13
   }
}
```

The above code gives an error when compiled. The reason for this is that when the subclass overrides a method of the super class, the method definition in the subclass can only specify all or a subset of the exception classes in the throws clause of the overridden method in the superclass.

- Overriding method throws runtime exception allowed.
- Overriding method throws subclass of the exception type declared by the parent method allowed.
- Overriding method throws base class of exception type declared by the parent method not allowed.
- Overriding method throws totally unrelated compile time exception replacing parent method throws clause not allowed.

9.5 User Defined Exception

Each application have specific constraints and available java exception class and its subclass can't meet these constraints. So there is a need to create user's own customized exception to address these constraints and ensure the integrity in the application.

There could be a exception which cannot reach the user as it looks, instead it could be replaced by some other exceptions. In this case, the only solutions is to write our own exception and show the details regarding that exception.

This can be done by extending the class Exception. The keyword "throw" is used to create a new Exception and throw it to the catch block. The keywords used in java application are try, catch and finally are used in implementing used-defined exceptions. This Exception class inherits all the method from Throwable class.

Declaring you own Exception:

You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes:

- All exceptions must be a child of Throwable.
- If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.
- If you want to write a runtime exception, you need to extend the RuntimeException class.

You can define our own Exception class as below:

```
class MyException extends Exception
{
}
```

You just need to extend the Exception class to create your own Exception class. These are considered to be checked exceptions.

you create own customized exception as per requirements of the application.

```
class MyException extends Exception{
   int a;
   MyException(int b) {
     a=b;
   }
   public String toString(){
     return ("Exception Number =  "+a) ;
  }
}
class JavaException{
   public static void main(String args[]){
  try{
       throw new MyException(2);
       // throw is used to create a new exception and throw it.
  }
 catch(MyException e){
    System.out.println(e) ;}}}
```

Throwing User defined Exception

As another exception in user defined exception also the "throw" statement is used to signal the occurrence of the exception within try block.

```
public class MyException extends Exception
    {
     String msg = "";
     int marks;
    public MyException()
         {
     }
    public MyException(String str)
         {
         super(str);
     }
    public String toString()
         {
         if(marks <= 40)
        msg = "You have failed";
         if(marks > 40)
        msg = "You have Passed";
        return msg;
     }
}
public class test
    {
    public static void main(String args[])
         {
        test t = new test();
         t.dd();
     }
    public void dd()
         {
         try
              {
             int i=0;
             if( i < 40)
           throw new MyException();
          }
        catch(MyException ee1)
              {
          System.out.println("my ex"+ee1);
          }
     }
}
```

9.6 Rethrowing Exception

When you catch an exception, it's possible to rethrow it. This is just the same as if you hadn't caught it in the first place - the exception will continue to bubble up through the layers until it reaches some other code that catches it. It means you have temporary access to the exception at the point where you caught it. You can log the fact that an error has occurred, but the real error handling is happening at a higher level.

Sometimes it is useful in a method to catch an exception that has been thrown, process it a bit in the method itself, and then rethrow the same (or a different) exception. Note that writing

```
        catch (RuntimeException re)
   {
System.out.println("Runtime exception: " + re.getMessage();
    throw re;
   }
```

tells Java to catch any RuntimeException (or one of its subclasses), print an error message, and then rethrow the same exception; here the throw statement indicates to throw not a new exception, but whatever exception object re now refers to.

If you might rethrow the IOException you catch, then you must declare it!"

Just as you can throw a new exception from a catch clause, you can also throw the same exception you just caught. Here's a catch clause that does this:

```
catch(IOException e) {
// Do things, then if you decide you can't handle it...
throw e;
}
```

Rethrowing an exception causes the exception to go to the exception handlers in the next-higher context. Any further catch clauses for the same try block are still ignored. In addition, everything about the exception object is preserved, so the handler at the higher context that catches the specific exception type can extract all the information from that object.

If you simply re-throw the current exception, the information that you print about that exception in printStackTrace( ) will pertain to the exception's origin, not the place where you re-throw it. If you want to install new stack trace information, you can do so by calling fillInStackTrace( ), which returns an exception object that it creates by stuffing the current stack information into the old exception object.

Example of use of rethrow

```
//: RethrowNew.java
// Rethrow a different object from the one that
// was caught
public class RethrowNew {
  public static void f() throws Exception {
    System.out.println(
      "originating the exception in f()");
    throw new Exception("thrown from f()");
  }
  public static void main(String[] args) {
    try {
      f();
    } catch(Exception e) {
      System.out.println(
        "Caught in main, e.printStackTrace()");
     e.printStackTrace();
    throw new NullPointerException("from main");
    }
  }
}
```

All other catch clauses associated with the same try are ignored, if a finally block exists, it runs, and the exception is thrown back to the calling method (the next method down the call stack). If you throw a checked exception from a catch clause, you must also declare that exception! In other words, you must handle and declare, as opposed to handle or declare. The following example is illegal:

```
public void doSomething() {
try {
// risky IO things
} catch(IOException ex) {
// can't handle it
throw ex; // Can't throw it unless you declare it
}
}
```

In the preceding code, the doSomething() method is clearly able to throw a checked exception-in this case an IOException-so the compiler says, "Well, you have a try/catch in there, but it's not good enough. If you might rethrow the IOException you catch, then you must declare it!"

Sometimes it is useful in a method to catch an exception that has been thrown, process it a bit in the method itself, and then rethrow the same (or a different) exception. Note that writing

```
          catch (RuntimeException re)
   {
System.out.println("Runtime exception: " + re.getMessage();
     throw re;
   }
```

tells Java to catch any RuntimeException (or one of its subclasses), print an error message, and then rethrow the same exception; here the throw statement indicates to throw not a new exception, but whatever exception object re now refers to.

9.7 Exception Propagation

There's no requirement that you code a catch clause for every possible exception that could be thrown from the corresponding try block. If a method doesn't provide a catch clause for a particular exception, that method is said to be Propagating" the exception.

An exception is first thrown from the top of the stack, and if it isn't caught by the catch of try who threw it. It drops down the call stack to the previous method, If not caught there, the exception again drops down to the previous method, and so on until it is caught or until it reaches the very bottom of the call stack. This is called exception propagation.

An exception that's never caught will cause your application to stop running. A description of the exception will be displayed, and the call stack will be dumped." This helps you debug your application by telling you what exception was thrown, from what method it was thrown, and what the stack looked like at the time. So make sure you handle all exceptions in your classes somewhere in the method calling hierarchy.

9.8 Wrapping Exception

To hide and separate the implementation details from the exposed interface in order to reduce the complexity and to simplify the interface.

- To wrap the exceptions in such a way that the original exception object is not used and only the message as String is encapsulated in the new exception.
- Create another exception that contains references to the thrown exception

In order to prevent such a change from propagating to higher layers, one can wrap such low level Exceptions in a generic "data exception" wrapper, designed exclusively to protect the higher layers from such changes.

Throwable.getCause() can always be used to extract the original exception, if desired. (However, you sometimes need to be careful that both client and server know about the class of an underlying exception.)

For example a client invokes remotely your api using rmi. The api throws the exception wrapped but the client have only the wrapper exception class in the classpath, not the wrapped one. An exception will be thrown. In cases like this one runtime exception should be created containing a descriptive message and should not wrap the original exception.

The main reason exception wrapping is used to prevent the code further up the call stack from having to know about every possible exception in the system. There are two main reasons for this.

The first reason is, that declared exceptions aggregate towards the top of the call stack. If you do not wrap exceptions, but instead pass them on by declaring your methods to throw them, you may end up with top level methods that declare many different exceptions. Declaring all these exceptions in each method back up the call stack becomes tedious.

The second reason is that you may not want your top level components to know anything about the bottom level components, nor the exceptions they throw.

9.9 Exception Enrichment

Exception enrichment is an alternative to exception wrapping. Exception wrapping has a couple of disadvantages that exception enrichment can fix. These disadvantages are:

- Exception wrapping may result in very long stack traces consisting of one stack trace for each exception in the wrapping hierarchy. Most often only the root stack trace is interesting. The rest of the stack traces are then just annoying.

The messages of the exceptions are spread out over the stack traces. The message of an exception is typically printed above the stack trace. When several exceptions wrap each other in a hierarchy, all these messages are spread out in between the stack traces. This makes it harder to determine what went wrong, and what the program was trying to do when the error happened. In other words, it makes it hard to determine in what context the error occurred.

Here is an example:

```
public void method2() throws EnrichableException{
  try{
    method1();
  } catch(EnrichableException e){
    e.addInfo("An error occurred when trying to ...");
```

```
        throw e;
      }
   }
 public void method1() throws EnrichableException {
    if(...) throw new EnrichableException(
       "Original error message");
   }
```

As you can see the method1() throws an EnrichableException which is a superclass for enrichable exceptions. This is not a standard Java exception, so you will have to create it yourself.

Notice how method2() calls the addInfo() method on the caught EnrichableException, and rethrow it afterwards. As the exception propagates up the call stack, each catch block can add relevant information to the exception if necessary.

Using this simple technique you only get a single stack trace, and still get any relevant contextual information necessary to investigate the cause of the exception.

Wrapping Non-Enrichable Exceptions

You may not always be able to avoid exception wrapping. If a component in your application throws a checked exception that is not enrichable, you may have to wrap it in an enrichable exception. Here is an example where method1() catches a non-enrichable exception and wraps it in an enrichable exception, and throws the enrichable exception:

```
  public void method1() throws EnrichableException {
    try{
        ... call some method that throws an IOException ...
     } catch(IOException ioexception)
        throw new EnrichableException( ioexception,
          "METHOD1", "ERROR1", "Original error message");
   }
```

9.10 Summary

Exceptions are long-established programming structures that allow programmer to handle multiple runtime errors through a centralized mechanism within client code.

Exception handling allows developers to detect errors easily without writing special code to test return values.

Checked exceptions must be explicitly caught or propagated. Unchecked exceptions do not have this requirement. Checked and unchecked exceptions are functionally equivalent. There is nothing you can do with checked exceptions that cannot also be done with unchecked exceptions, and vice versa. None is functionally better than the other.

Using inheritance with exceptions enables an exception handler to catch related errors with concise notation. One approach is to catch each type of pointer or reference to a derived-class exception object individually, but a more concise approach is to catch pointers or references to base-class exception objects instead.

Also, catching pointers or references to derived-class exception objects individually is error prone, especially if you forget to test explicitly for one or more of the derived-class pointer or reference types

You can create your own exceptions in Java. "throw" is used to create a new Exception and throw it to the catch block.

Sometimes it is useful in a method to catch an exception that has been thrown, process it a bit in the method itself, and then rethrow the same (or a different) exception.

There's no requirement that you code a catch clause for every possible exception that could be thrown from the corresponding try block. If a method doesn't provide a catch clause for a particular exception, that method is said to be Propagating" the exception.

In order to prevent such a change from propagating to higher layers, one can wrap such low level Exceptions in a generic "data exception" wrapper, designed exclusively to protect the higher layers from such changes.

Exception enrichment is an alternative to exception wrapping. Exception wrapping has a couple of disadvantages that exception enrichment can fix.

9.11 Self-Assessment Questions

1. Differentiate between checked and unchecked exception.
2. Explain significance of User defined exception.
3. How you can declare your own exception.
4. Write Short note on - Rethrowing and propagation of exception.
5. What do you mean by exception wrapping and exception enrichment.

9.12 References

1. Programming with Java A Primer, E.Balaguruswamy Tata McGraw Hill Companies
2. Java Programming John P. Flynt Thomson 2nd
3. Java Programming Language Ken Arnold Pearson
4. The complete reference JAVA2, Herbert schildt. TMH
5. Big Java, Cay Horstmann 2nd edition, Wiley India Edition
6. Core Java, Dietel and Dietel
7. Java - Balaguruswamy
8. Java server programming, Ivan Bayross SPD

Unit - 10
Multithreading

Structure of the Unit

10.0 Objective

This chapter provides a general overview of

- Threading
- Thread life cycle
- Thread priorities and thread scheduling
- Thread synchronization
- Thread communication

10.1 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.

A multithreading is a specialized form of multitasking. Multitasking threads require less overhead than multitasking processes.

There is one important term with threads which is process, a process consists of the memory space allocated by the operating system that can contain one or more threads. A thread cannot exist on its own; it must be a part of a process. A process remains running until all of the non-daemon threads are done executing.

Multithreading enables us to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.

10.2 Life Cycle of a Thread

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. Following diagram shows complete life cycle of a thread.

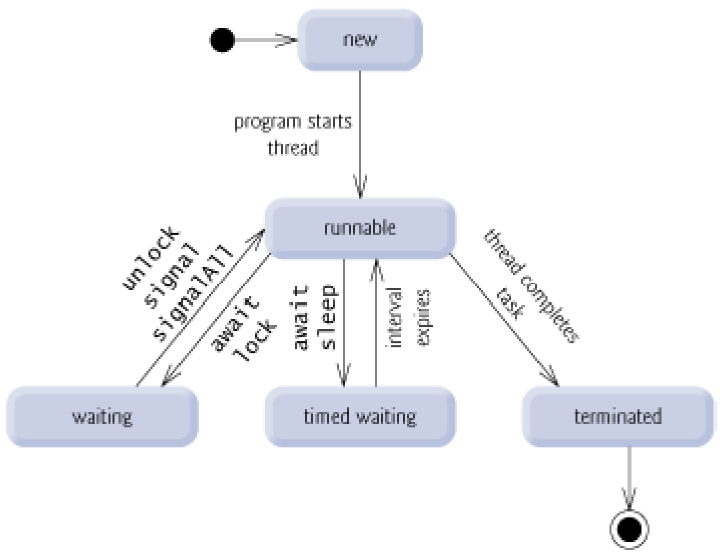


Figure 10.1 : Life cycle of a thread

Above mentioned stages are explained here:

- New: A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
- Runnable: After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- Waiting: Sometimes a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- Timed waiting: A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- Terminated: A runnable thread enters the terminated state when it completes its task or otherwise terminates.

10.3 Thread Priorities and Scheduling

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependent.

When a thread is created, it inherits its priority from the thread that created it. In addition, byusing the setPriority method, thread's priority can be modified at any time after its creation. Thread priorities are integers that range between MIN_PRIORITY and MAX_PRIORITY (constants defined in the Thread class). The higher the integer, the higher the priority. At any given time, when multiple threads are ready to be executed, the runtime system chooses the Runnable thread for execution that has the highest priority. Only when that thread stops, yields, or becomes Not Runnable will a lower-priority thread start executing. If two threads of the same priority are waiting for the CPU, the scheduler arbitrarily chooses one of them to run first. The chosen thread runs until one of the following conditions is true:

- A higher-priority thread becomes runnable.
- The thread yields, or its run method exits.
- On systems that support time-slicing, the thread's time allotment has expired.

Then the second thread is given a chance to run, and so on, until the interpreter exits.

The Java runtime system's thread-scheduling algorithm is also preemptive. If at any time a thread with a higher priority than all other Runnable threads becomes Runnable, the runtime system chooses the new higher-priority thread for execution. The new thread is said to preempt the other threads.

10.4 Creation of a Thread

Java defines two ways in which a thread can be created:

- Using Runnable interface.
- By extending the Thread class, itself.

Create Thread by Implementing Runnable:

The easiest way to create a thread is to create a class that implements the Runnable interface. To implement Runnable, a class needs only implement a single method called run( ), which is declared like this:

Public void run()

We can define the code that constitutes the new thread inside run() method. It is important to understand that run() can call other methods, use other classes, and declare variables, just like the main thread can.

After creating a class that implements Runnable, we can instantiate an object of type Thread from within that class. Thread defines several constructors. The one that we will use is shown here:

Thread(Runnable threadOb, String threadName);

Here threadOb is an instance of a class that implements the Runnable interface and the name of the new thread is specified by threadName.

After the new thread is created, it will not start running until you call its start( ) method, which is declared within Thread. The start( ) method is shown here:

void start( );

Here is an example that creates a new thread and starts it's running:

```
// Create a new thread.
class NewThread implements Runnable {
  Thread t;
  NewThread() {
     // Create a new, second thread
    t = new Thread(this, "Demo Thread");
    System.out.println("Child thread: " + t);
    t.start(); // Start the thread
   }
  // This is the entry point for the second thread.
  public void run() {
      try {
         for(int i = 5; i > 0; i--) {
          System.out.println("Child Thread: " + i);
           // Let the thread sleep for a while.
            Thread.sleep(500);
          }
    } catch (InterruptedException e) {
       System.out.println("Child interrupted.");
     }
    System.out.println("Exiting child thread.");
   }
}
class ThreadDemo {
  public static void main(String args[]) {
     new NewThread(); // create a new thread
      try {
         for(int i = 5; i > 0; i--) {
         System.out.println("Main Thread: " + i);
           Thread.sleep(1000);
          }
     } catch (InterruptedException e) {
       System.out.println("Main thread interrupted.");
      }
    System.out.println("Main thread exiting.");
   }
}
```

This would produce following result:

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

Create Thread by Extending Thread:

The other way to create a thread is to create a new class that extends Thread, and then to create an instance of that class.

The extending class must override the run( ) method, which is the entry point for the new thread. It must also call start( ) to begin execution of the new thread.

Here is the preceding program rewritten to extend Thread:

```
// Create a second thread by extending Thread
class NewThread extends Thread {
  NewThread() {
     // Create a new, second thread
    super("Demo Thread");
    System.out.println("Child thread: " + this);
    start(); // Start the thread
   }
  // This is the entry point for the second thread.
  public void run() {
     try {
       for(int i = 5; i > 0; i--) {
        System.out.println("Child Thread: " + i);
                              // Let the thread sleep for a while.
          Thread.sleep(500);
        }
    } catch (InterruptedException e) {
      System.out.println("Child interrupted.");
     }
   System.out.println("Exiting child thread.");
   }
}
class ExtendThread {
  public static void main(String args[]) {
    new NewThread(); // create a new thread
     try {
       for(int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
          Thread.sleep(1000);
        }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted.");
     }
    System.out.println("Main thread exiting.");
   }
}
```

This would produce following result:

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

10.5 Thread Methods

Following is the list of important methods available in the Thread class.

SN	Methods with Description
1	public void start() Starts the thread in a separate path of execution, then invokes the run() method on this Thread object.
2	public void run() If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.
3	public final void setName(String name) Changes the name of the Thread object. There is also a getName() method for retrieving the name.
4	public final void setPriority(int priority) Sets the priority of this Thread object. The possible values are between 1 and 10.
5	public final void setDaemon(boolean on) A parameter of true denotes this Thread as a daemon thread.
6	public final void join(long millisec) The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes.
7	public void interrupt() Interrupts this thread, causing it to continue execution if it was blocked for any reason.
8	public final boolean isAlive() Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.

The previous methods are invoked on a particular Thread object. The following methods in the Thread class are static. Invoking one of the static methods performs the operation on the currently running thread.

SN	Methods with Description
1	public static void yield() Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled
2	public static void sleep(long millisec) Causes the currently running thread to block for at least the specified number of milliseconds
3	public static boolean holdsLock(Object x) Returns true if the current thread holds the lock on the given Object.
4	public static Thread currentThread() Returns a reference to the currently running thread, which is the thread that invokes this method.
5	public static void dumpStack() Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application.

Example:

The following ThreadClassDemo program demonstrates some of these methods of the Thread class:

```
// File Name : DisplayMessage.java
// Create a thread to implement Runnable
public class DisplayMessage implements Runnable
{
  private String message;
  public DisplayMessage(String message)
  {
    this.message = message;
  }
  public void run()
  {
    while(true)
    {
      System.out.println(message);
    }
  }
}
// File Name : GuessANumber.java
// Create a thread to extentd Thread
public class GuessANumber extends Thread
{
  private int number;
  public GuessANumber(int number)
   {
     this.number = number;
   }
  public void run()
   {
     int counter = 0;
     int guess = 0;
      do
      {
        guess = (int) (Math.random() * 100 + 1);
       System.out.println(this.getName()
                    + " guesses " + guess);
         counter++;
     }while(guess != number);
     System.out.println("** Correct! " + this.getName()
                   + " in " + counter + " guesses.**");
   }
}
// File Name : ThreadClassDemo.java
public class ThreadClassDemo
{
  public static void main(String [] args)
   {
     Runnable hello = new DisplayMessage("Hello");
     Thread thread1 = new Thread(hello);
     thread1.setDaemon(true);
     thread1.setName("hello");
     System.out.println("Starting hello thread...");
     thread1.start();
     Runnable bye = new DisplayMessage("Goodbye");
     Thread thread2 = new Thread(hello);
     thread2.setPriority(Thread.MIN_PRIORITY);
     thread2.setDaemon(true);
     System.out.println("Starting goodbye thread...");
     thread2.start();
     System.out.println("Starting thread3...");
     Thread thread3 = new GuessANumber(27);
     thread3.start();
      try
```

```
    {
      thread3.join();
   }catch(InterruptedException e)
    {
     System.out.println("Thread interrupted.");
    }
   System.out.println("Starting thread4...");
   Thread thread4 = new GuessANumber(75);

         thread4.start();
   System.out.println("main() is ending...");
  }
}
```

This would produce following result. You can try this example again and again and you would get different result every time.

```
Starting hello thread...
Starting goodbye thread...
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Thread-2 guesses 27
Hello
** Correct! Thread-2 in 102 guesses.**
Hello
Starting thread4...
Hello
Hello
..........remaining result produced.
```

10.6 Thread Synchronization

The fact that a thread can execute in parallel with other threads is the main power of the concurrent programming. But this property causes usually a lot of trouble for programmers which must assure a correct sharing of the resources used in common by different threads. It means that the threads that execute in parallel and use common critical resources must be synchronized at some points to assure a correct functioning of the system. The correct functioning of a parallel or concurrent program is possible only if the conflicts of simultaneously performing a critical section operation by several threads are avoided. A critical section operation is an operation (a portion of the program code) that can not be done in parallel by several processes or treads, like incrementing a sheared counter, writing in a shared buffer, modifying a shared object, etc.

The synchronization means that a thread must wait for another thread to leave the critical section and to enter into this section using some security measures, e.g. locking the critical section when entering it.

Java uses word synchronized to synchronize threads and intercommunicate to each other. It is basically a mechanism which allows two or more threads to share all the available resources in a sequential manner. Java's synchronized is used to ensure that only one thread is in a critical region. critical region is a lock area where only one thread is run (or lock) at a time. Once the thread is in its critical section, no other thread can enter to that critical region. In that case, another thread will has to wait until the current thread leaves its critical section.

General form of the synchronized statement is as:

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

Lock term refers to the access granted to a particular thread that can access the shared resources. At any given time, only one thread can hold the lock and thereby have access to the shared resource. Every object in Java has build-in lock that only comes in action when the object has synchronized method code. By associating a shared resource with a Java object and its lock, the object can act as a guard, ensuring synchronized access to the resource.

Only one thread at a time can access the shared resource guarded by the object lock.

Since there is one lock per object, if one thread has acquired the lock, no other thread can acquire the lock until the lock is not released by first thread. Acquire the lock means the thread currently in synchronized method and released the lock means exits the synchronized method.

Remember the following points related to lock and synchronization:

- Only methods (or blocks) can be synchronized, Classes and variable cannot be synchronized.
- Each object has just one lock.
- All methods in a class need not to be synchronized. A class can have both synchronized and non-synchronized methods.
- If two threads wants to execute a synchronized method in a class, and both threads are using the same instance of the class to invoke the method then only one thread can execute the method at a time.
- If a class has both synchronized and non-synchronized methods, multiple threads can still access the class's non-synchronized methods. If you have methods that don't access the data you're trying to protect, then you don't need to synchronize them. Synchronization can cause a hit in some cases (or even deadlock if used incorrectly), so you should be careful not to overuse it.
- If a thread goes to sleep, it holds any locks it has? it doesn't release them.
- A thread can acquire more than one lock. For example, a thread can enter a synchronized method, thus acquiring a lock, and then immediately invoke a synchronized method on a different object, thus acquiring that lock as well. As the stack unwinds, locks are released again.
- You can synchronize a block of code rather than a method.
- Constructors cannot be synchronized

There are two ways to synchronized the execution of code:

1. Synchronized Methods
2. Synchronized Blocks (Statements)

1. Synchronized Methods:

Any method is specified with the keyword synchronized is only executed by one thread at a time. If any thread want to execute the synchronized method, firstly it has to obtain the objects lock. If the lock is already held by another thread, then calling thread has to wait.

Synchronized methods are useful in those situations where methods are executed concurrently, so that these can be intercommunicate manipulate the state of an object in ways that can corrupt the state if . Stack implementations usually define the two operations push and pop of elements as synchronized, pushing and popping are mutually exclusive operations. For Example if several threads were sharing a stack, if one thread is popping the element on the stack then another thread would not be able to pushing the element on the stack.

Program for synchronized thread.

```
class Share extends Thread{
 static String msg[]={"This", "is", "a", "synchronized", "variable"};
 Share(String threadname){
 super(threadname);
  }
 public void run(){
 display(getName());
  }
 public synchronized void display(String threadN){
 for(int i=0;i<=4;i++)
 System.out.println(threadN+msg[i]);
 try{
 this.sleep(1000);
 }catch(Exception e){}
  }
}
public class SynThread1 {
 public static void main(String[] args) {
 Share t1=new Share("Thread One: ");
 t1.start();
 Share t2=new Share("Thread Two: ");
 t2.start();
}
}
```

Output of the program is:

```
C:\>javac SynThread.java
C:\>java SynThread
Thread One: This
Thread One: is
Thread One: a
Thread One: synchronized
Thread One: variable
Thread Two: This
Thread Two: is
Thread two: a
Thread Two: synchronized
Thread Two: variable
```

In this program, the method "display( )" is synchronized that will be shared by both thread's objects at the time of program execution. Thus only one thread can access that method and process it until all statements of the method are executed.

Synchronized Blocks (Statements)

Other way of handling synchronization is Synchronized Blocks (Statements). Synchronized statements must specify the object that provides the native lock. The synchronized block allows execution of arbitrary code to be synchronized on the lock of an arbitrary object.

General form of synchronized block is:

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

The following program demonstrates the synchronized block that shows the same output as the output of the previous example:

```
class Share extends Thread{
 static String msg[]={"This", "is", "a", "synchronized", "variable"};
 Share(String threadname){
 super(threadname);
  }
 public void run(){
 display(getName());
  }
 public void display(String threadN){
  synchronized(this){
 for(int i=0;i<=4;i++)
 System.out.println(threadN+msg[i]);
 try{
 this.sleep(1000);
 }catch(Exception e){}
  }
}
public class SynStatement {
 public static void main(String[] args) {
 Share t1=new Share("Thread One: ");
 t1.start();
 Share t2=new Share("Thread Two: ");
 t2.start();
}
}
```

Output of the Program

```
 C:\>javac SynStatement.java
C:\>java SynStatement
Thread One: This
Thread One: is
Thread One: a
Thread One: synchronized
Thread One: variable
Thread Two: This
Thread Two: is
Thread Two: a
Thread Two: synchronized
Thread Two: variable
```

10.7 Thread Groups

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

The runtime system puts a thread into a thread group during thread construction. When you create a thread, you can either allow the runtime system to put the new thread in some reasonable default group or you can explicitly set the new thread's group. The thread is a permanent member of whatever thread group it joins upon its creation--you cannot move a thread to a new group after the thread has been created.

The Default Thread Group

If we create a new Thread without specifying its group in the constructor, the runtime system automatically places the new thread in the same group as the thread that created it (known as the current thread group and the current thread, respectively). So, if you leave the thread group unspecified when you create your thread, what group contains your thread?

When a Java application first starts up, the Java runtime system creates a ThreadGroup named main. Unless specified otherwise, all new threads that you create become members of the main thread group.

Many Java programmers ignore thread groups altogether and allow the runtime system to handle all of the details regarding thread groups. However, if your program creates a lot of threads that should be manipulated as a group, or if you are implementing a custom security manager, you will likely want more control over thread groups. Continue reading for more details!

Creating a Thread Explicitly in a Group

As mentioned previously, a thread is a permanent member of whatever thread group it joins when its created--you cannot move a thread to a new group after the thread has been created. Thus, if you wish to put your new thread in a thread group other than the default, you must specify the thread group explicitly when you create the thread. The Thread class has three constructors that let you set a new thread's group:

public Thread(ThreadGroup group, Runnable target)

public Thread(ThreadGroup group, String name)

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

Each of these constructors creates a new thread, initializes it based on the Runnable and String parameters, and makes the new thread a member of the specified group. For example, the following code sample creates a thread group (myThreadGroup) and then creates a thread (myThread) in that group.

ThreadGroup myThreadGroup = new ThreadGroup("My Group of Threads");

Thread myThread = new Thread(myThreadGroup, "a thread for my group");

The ThreadGroup passed into a Thread constructor does not necessarily have to be a group that you create--it can be a group created by the Java runtime system, or a group created by the application in which your applet is running.

Getting a Thread's Group

To find out what group a thread is in, we can call its getThreadGroup method:

theGroup = myThread.getThreadGroup();

The ThreadGroup Class

Once we have obtained a thread's ThreadGroup, we can query the group for information, such as what other threads are in the group. We can also modify the threads in that group, such as suspending, resuming, or stopping them, with a single method invocation.

10.8 Communication of Threads

Java provides a very efficient way through which multiple-threads can communicate with each-other. This way reduces the CPU's idle time i.e. A process where, a thread is paused running in its critical region and another thread is allowed to enter (or lock) in the same critical section to be executed. This technique is known as Interthread communication which is implemented by some methods. These methods are defined in "java.lang" package and can only be called within synchronized code shown as:

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

All these methods must be called within a try-catch block.

Lets see an example implementing these methods :

```
class Shared {
int num=0;
boolean value = false;
synchronized int get() {
  if (value==false)
  try {
 wait();
```

```
  }
 catch (InterruptedException e) {
 System.out.println("InterruptedException caught");
  }
System.out.println("consume: " + num);
value=false;
notify();
return num;
}
synchronized void put(int num) {
 if (value==true)
 try {
 wait();
  }
 catch (InterruptedException e) {
 System.out.println("InterruptedException caught");
  }
 this.num=num;
 System.out.println("Produce: " + num);
 value=false;
 notify();
  }
  }
 class Producer extends Thread {
  Shared s;
 Producer(Shared s) {
 this.s=s;
 this.start();
  }
 public void run() {
 int i=0;
 s.put(++i);
  }
}
class Consumer extends Thread{
  Shared s;
 Consumer(Shared s) {
 this.s=s;
 this.start();
  }
 public void run() {
 s.get();
```

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

Output of the Program:

```
C:\>javac InterThread.java
C:\>java InterThread
Produce:1
consume: 1
```

10.9 Summary

•A multithreading is a specialized form of multitasking. Multitasking threads require less overhead than multitasking processes.

•Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

•Java uses word synchronized to synchronize threads and intercommunicate to each other.

•Every Java thread is a member of a thread group. Thread groups provide a mechanism for collecting multiple threads into a single object and manipulating those threads all at once, rather than individually.

•Java provides a very efficient way through which multiple-threads can communicate with each-other. This way reduces the CPU's idle time.

10.10 Self-Assessment Questions

1. Describe the life cycle of a thread.
2. Describe some methods which can be used by Threads.
3. What are the different methods of thread creation? Explain.
4. What is the need of thread groups?
5. What is interthreadcommunication?

10.11 References

1. Programming with Java A Primer, E.Balaguruswamy Tata McGraw Hill Companies
2. Java Programming John P. Flynt Thomson 2nd
3. Java Programming Language Ken Arnold Pearson
4. The complete reference JAVA2, Herbert schildt. TMH
5. Big Java, Cay Horstmann 2nd edition, Wiley India Edition
6. Core Java, Dietel and Dietel
7. Java - Balaguruswamy
8. Java server programming, Ivan Bayross SPD

Unit - 11

File Input and Output

Structure of the Unit

11.0 Objective

This chapter provides a general overview of

- Streams
- Implementation of the stream for reading data from console
- Implementation of the stream for writing data to console
- I/O operations on files

11.1 Introduction

Java provides strong, flexible support for I/O as it relates to files and networks. This chapter covers the Java platform classes which are used for basic I/O. It first discusses I/O Streams, a powerful concept that greatly simplifies I/O operations.

Most of the classes covered in the I/O Streams section are in the java.io package. Most of the classes covered in the File I/O section are in the java.nio.file package.

A stream is a sequence of data. The InputStream is used to read data from a source and the OutputStream is used for writing data to a destination.

11.2 System Class

The System class provides facilities like standard input, output and error streams. It also provides means to access properties associated with the Java runtime system and various environment properties such as version, path, vendor and so on. Fields of this class are in, out and err which represent the standard input, output and error respectively.

The three streams System.in, System.out, and System.err are also common sources or destinations of data. Most commonly used is System.out for writing output to the console.

These three streams are initialized by the Java runtime when a JVM starts up, so we don't have to instantiate any streams by ourself.

11.2.1 System.in

System.in is an InputStream which is typically connected to keyboard input of console programs. System.in is not used as often since data is commonly passed to a command line Java application via command line arguments, or configuration files. In applications with GUI the input to the application is given via the GUI. This is a separate input mechanism from Java IO.

11.2.2 System.out

System.out is a PrintStream. System.out normally outputs the data we write to it to the console. This is often used from console-only programs like command line tools. This is also often used to print debug statements of from a program.

11.2.3 System.err

System.err is a PrintStream. System.err works like System.out except it is normally only used to output error texts. Some programs (like Eclipse) will show the output to System.err in red text, to make it more obvious that it is error text.

Here is a simple example that uses System.out and System.err:

```
try {
 InputStream input = new FileInputStream("c:\\data\\...");
 System.out.println("File opened...");
} catch (IOException e){
 System.err.println("File opening failed:");
 e.printStackTrace();
}
```

It must be noted that the System class cannot be instantiated to create objects, or in simple words you cannot create objects out of the System class.

The following program retrieves and displays some of the Java related environment properties.

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

```
  {
  System.out.println("The Java class path is : "+System.getProperty("java.class.path"));
    System.out.println("The Java installation directory is : "+System.getProperty("java.home"));
  System.out.println("The Java class version number is : "+System.getProperty("java.class.version"));
    System.out.println("The Java vendor specification is :
"+System.getProperty("java.specification.vendor"));
    System.out.println("The Java specified version is :
"+System.getProperty("java.specification.version"));
    System.out.println("The Java vendor is : "+System.getProperty("java.vendor"));
    System.out.println("The Java vendor's website is : "+System.getProperty("java.vendor.url"));
    System.out.println("The Java jdk version is : "+System.getProperty("java.version"));
    System.out.println("The Java virtual machine name is : "+System.getProperty("java.vm.name"));
  }
}
```

Each property to be printed is supplied as a String parameter to the System.getProperty() method. This method in turn will return the relevant information to the System.out.println() method. The getProperty() method returns the value associated with it. If there is no value then it returns null. In getProperty() method, a second argument can also be specified which can be user defined. So instead of returning a null value, the default value mentioned as the second argument will be returned and the risk of NullPointer exception can be avoided.

11.3 The "Class" class

Instances of this class encapsulate the run time state of an object in a running Java application. This allows retrieving info about the object during runtime.

An object of this class or an instance can be obtained in one of the three ways:

o Use getClass( ) method in an object.

o Use static forName( ) method of Class to get an instance of Class using the name of the class.

o Load a new class using a custom classLoader object.

It must be noted that there is no public constructor for Class.The code in the next example illustrates how relevant info of a class can be retrieved using methods of class.

```
interface A
{
  final int id = 1;
  final String name = "Pencil";
}
class B implements A
{
  int deptno;
}
```

```
class ClassDemo
{
   public static void main(String args[])
    {
        A Obja = new B();
        B Objb = new B();
        Class Objx;
        Objx = Obja.getClass();
      System.out.println("Obja is object of type: "+ Objx.getName());
        Objx = Objb.getClass();
      System.out.println("Objb is object of type: "+ Objx.getName());
        Objx = Objx.getSuperclass();
      System.out.println("Objb's superclass is : "+ Objx.getName());
    }
}
```

11.4 I/O Stream

An I/O Stream represents an input source or an output destination. A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and arrays.

Streams support different kinds of data, including simple bytes, primitive data types, localized characters, and objects. Some streams simply pass on data; some manipulate and transform the data in useful ways. No matter how they work internally, all streams present the same simple model to programs that use them: A stream is a sequence of data. A program uses an input stream to read data from a source, one item at a time:

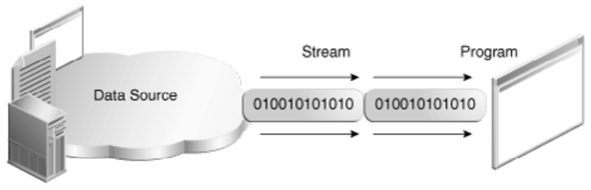


Figure 11.1 Reading information into a program.

A program uses an output stream to write data to a destination, one item at a time.

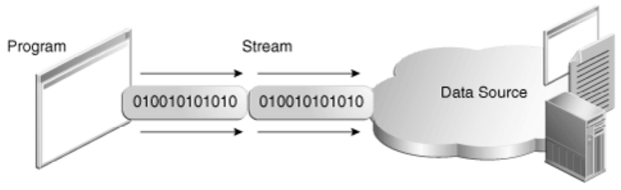


Figure 11.2 Writing information from a program.

The data source and data destination pictured above can be anything that holds, generates, or consumes data. Obviously this includes disk files, but a source or destination can also be another program, a peripheral device, a network socket, or an array.

11.4.1 Byte Stream

Programs use byte streams to perform input and output of 8-bit bytes. All byte stream classes are descended from InputStream and OutputStream.

There are many byte stream classes. To demonstrate how byte streams work, we will focus on the file I/O byte streams, FileInputStream and FileOutputStream. Other kinds of byte streams are used in much the same way; they differ mainly in the way they are constructed.

Using Byte Streams

FileInputStream and FileOutputStream can be explored by examining an example program named CopyBytes, which uses byte streams to copy abc.txt, one byte at a time.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class CopyBytes {
    public static void main(String[] args) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;
        try {
            in = new FileInputStream("abc.txt");
            out = new FileOutputStream("outagain.txt");
            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

CopyBytes spends most of its time in a simple loop that reads the input stream and writes the output stream, one byte at a time.

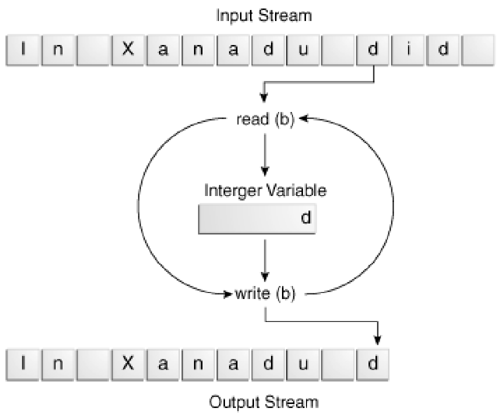


Figure 11.3 Simple byte stream input and output

Note that read() returns an int value. If the input is a stream of bytes, why doesn't read() return a byte value? Using a int as a return type allows read() to use -1 to indicate that it has reached the end of the stream.

Close Streams

Closing a stream when it is no longer needed is very important - so important that CopyBytes uses a finally block to guarantee that both streams will be closed even if an error occurs. This practice helps avoid serious resource leaks.

One possible error is that CopyBytes was unable to open one or both files. When that happens, the stream variable corresponding to the file never changes from its initial null value. That's why CopyBytes makes sure that each stream variable contains an object reference before invoking close.

11.4.2 Character Stream

The Java platform stores character values using Unicode conventions. Character stream I/O automatically translates this internal format to and from the local character set. In Western locales, the local character set is usually an 8-bit superset of ASCII.

For most applications, I/O with character streams is no more complicated than I/O with byte streams. Input and output done with stream classes automatically translates to and from the local character set. A program that uses character streams in place of byte streams automatically adapts to the local character set and is ready for internationalization - all without extra effort by the programmer.

Using Character Streams

All character stream classes are descended from Reader and Writer. As with byte streams, there are character stream classes that specialize in file I/O: FileReader and FileWriter. The CopyCharacters example illustrates these classes.

```
import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

public class CopyCharacters {

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

        FileReader inputStream = null;

        FileWriter outputStream = null;

        try {

            inputStream = new FileReader("xanadu.txt");

            outputStream = new FileWriter("characteroutput.txt");

            int c;

            while ((c = inputStream.read()) != -1) {

                outputStream.write(c);

            }

        } finally {

            if (inputStream != null) {
                inputStream.close();
            }
          if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

CopyCharacters is very similar to CopyBytes. The most important difference is that CopyCharacters uses FileReader and FileWriter for input and output in place of FileInputStream and FileOutputStream. Notice that both CopyBytes and CopyCharacters use an int variable to read to and write from. However, in CopyCharacters, the int variable holds a character value in its last 16 bits; in CopyBytes, the int variable holds a byte value in its last 8 bits.

Character Streams that Use Byte Streams

Character streams are often wrappers for byte streams. The character stream uses the byte stream to perform the physical I/O, while the character stream handles translation between characters and bytes. FileReader, for example, uses FileInputStream, while FileWriter uses FileOutputStream.

There are two general-purpose byte-to-character "bridge" streams: InputStreamReader and OutputStreamWriter. Use them to create character streams when there are no prepackaged character stream classes that meet your needs. The sockets lesson in the networking trail shows how to create character streams from the byte streams provided by socket classes.

Line-Oriented I/O

Character I/O usually occurs in bigger units than single characters. One common unit is the line: a string of characters with a line terminator at the end. A line terminator can be a carriage-return/line-feed sequence ("\r\n"), a single carriage-return ("\r"), or a single line-feed ("\n"). Supporting all possible line terminators allows programs to read text files created on any of the widely used operating systems.

Let's modify the CopyCharacters example to use line-oriented I/O. To do this, we have to use two classes we haven't seen before, BufferedReader and PrintWriter. We'll explore these classes in greater depth in Buffered I/O and Formatting. Right now, we're just interested in their support for line-oriented I/O.

The CopyLines example invokes BufferedReader.readLine and PrintWriter.println to do input and output one line at a time.

```
import java.io.FileReader;

import java.io.FileWriter;

import java.io.BufferedReader;

import java.io.PrintWriter;

import java.io.IOException;

public class CopyLines {
  public static void main(String[] args) throws IOException {
    BufferedReader inputStream = null;
    PrintWriter outputStream = null;
     try {
       inputStream = new BufferedReader(new FileReader("xanadu.txt"));
  outputStream = new PrintWriter(new FileWriter("characteroutput.txt"));
         String l;
       while ((l = inputStream.readLine()) != null) {
          outputStream.println(l);
          }
     } finally {
       if (inputStream != null) {
           inputStream.close();
          }
       if (outputStream != null) {
           outputStream.close();
          }
       }
   }
}
```

Invoking readLine returns a line of text with the line. CopyLines outputs each line using println, which appends the line terminator for the current operating system. This might not be the same line terminator that was used in the input file.

11.5 Reading Console Input

Java input console is accomplished by reading from System.in. To obtain a character-based stream that is attached to the console, we wrap System.in in a BufferedReader object, to create a character stream. Here is most common syntax to obtain BufferedReader:

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

Once BufferedReader is obtained, we can use read( ) method to reach a character or readLine( ) method to read a string from the console.

Reading Characters from Console:

To read a character from a BufferedReader, read( ) method is used, whose syntax is as follows:

int read( ) throws IOException

Each time that read( ) is called, it reads a character from the input stream and returns it as an integer value. It returns .1 when the end of the stream is encountered. It can throw an IOException.

The following program demonstrates read( ) by reading characters from the console until the user types a "r":

```
// Use a BufferedReader to read characters from the console.
import java.io.*;
class BRRead {
  public static void main(String args[]) throws IOException
   {
      char c;
     // Create a BufferedReader using System.in
     BufferedReader br = new BufferedReader(new
                    InputStreamReader(System.in));
     System.out.println("Enter characters, 'r' to quit.");
      // read characters
      do {
         c = (char) br.read();
       System.out.println(c);
      } while(c != 'r');
   }
}
```

Here is a sample run:

```
Enter characters, 'r' to quit.
321bcdr
3
2
1
b
c
d
r
```

Reading Strings from Console:

To read a string from the keyboard, use the version of readLine( ) that is a member of the BufferedReader class. Its general form is shown here:

```
String readLine( ) throws IOException
```

The following program demonstrates BufferedReader and the readLine( ) method. The program reads and displays lines of text until "e" is entered:

```
// Read a string from console using a BufferedReader.

import java.io.*;

class BRReadLines {

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

   {

    // Create a BufferedReader using System.in

    BufferedReader br = new BufferedReader(new

                         InputStreamReader(System.in));

    String str;
     System.out.println("Enter lines of text.");
     System.out.println("Enter 'e' to quit.");
     do {
        str = br.readLine();
        System.out.println(str);
     } while(!str.equals("e"));
   }
}
```

Here is a sample run:

```
Enter lines of text.
Enter 'e' to quit.
This is line one
This is line one
This is line two
This is line two
e

e
```

11.6 Writing Console Output

Console output is most easily accomplished with print( ) and println( ). These methods are defined by the class PrintStream which is the type of the object referenced by System.out. Even though System.out is a byte stream, using it for simple program output is still acceptable.

Because PrintStream is an output stream derived from OutputStream, it also implements the low-level method write( ). Thus, write( ) can be used to write to the console. The simplest form of write( ) defined by PrintStream is shown here:

```
void write(int byteval)
```

This method writes to the stream the byte specified by byteval. Although byteval is declared as an integer, only the low-order eight bits are written.

Here is a example that uses write( ) to output the character "R" followed by a newline to the screen:

```
import java.io.*;

// Demonstrate System.out.write().

class WriteDemo {
  public static void main(String args[]) {
     int b;
      b = 'R';
    System.out.write(b);
    System.out.write('\n');
   }
}
```

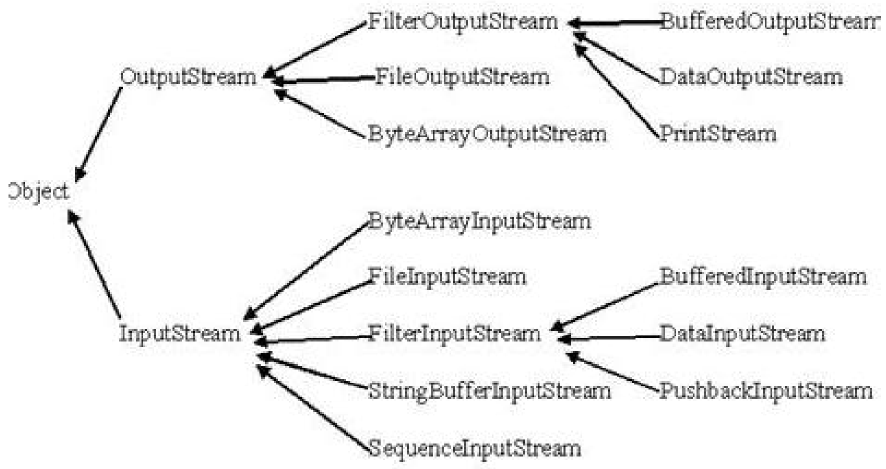
This would produce simply 'R' character on the output screen.

```
R
```

11.7 Reading and Writing Files

It is already discussed that a stream can be defined as a sequence of data. The InputStream is used to read data from a source and the OutputStream is used for writing data to a destination.

Here is a hierarchy of classes to deal with Input and Output streams.



The two important streams are FileInputStream and FileOutputStream which would be discussed in this tutorial:

11.7.1 FileInputStream

This stream is used for reading data from the files. Objects can be created using the keyword new and there are several types of constructors available.

Following constructor takes a file name as a string to create an input stream object to read the file.

```
InputStream f = new FileInputStream("C:/java/hello");
```

Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows:

```
File f = new File("C:/java/hello");
InputStream f = new FileInputStream(f);
```

Once we have InputStream object in hand then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

S.N.	Methods with Description
1	public void close() throws IOException{} This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	protected void finalize()throws IOException {}
	This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	public int read(int r)throws IOException{} This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's end of file.
4	public int read(byte[] r) throws IOException{} This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If end of file -1 will be returned.
5	public int available() throws IOException{} Gives the number of bytes that can be read from this file input stream. Returns an int.

The FileInputStream class makes it possible to read the contents of a file as a stream of bytes.

```
InputStream input = new FileInputStream("c:\\data\\input-text.txt");

int data = input.read();

while(data != -1) {

  //do something with data...

  doSomethingWithData(data);

  data = input.read();

}

input.close();
```

The read() method of a FileInputStream returns an int which contains the byte value of the byte read. If the read() method returns -1, there is no more data to read in the stream, and it can be closed. That is, -1 as int value, not -1 as byte value. There is a difference here!

The FileInputStream has other constructors too, letting you specify the file to be read in different ways.

11.7.2 FileOutputStream

FileOutputStream is used to create a file and write data into it.The stream would create a file, if it doesn't already exist, before opening it for output. Here are two constructors which can be used to create a FileOutputStream object.

Following constructor takes a file name as a string to create an input stream object to write the file.:

```
OutputStream f = new FileOutputStream("C:/java/hello")
```

Following constructor takes a file object to create an output stream object to write the file. First we create a file object using File() method as follows:

```
File f = new File("C:/java/hello");

OutputStream f = new FileOutputStream(f);
```

Once we have OutputStream object in hand then there is a list of helper methods which can be used to write to stream or to do other operations on the stream.

S.N.	Methods with Description
1	public void close() throws IOException{} This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	protected void finalize()throws IOException {} This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	public void write(int w)throws IOException{} This methods writes the specified byte to the output stream.
4	public void write(byte[] w) Writes w.length bytes from the mentioned byte array to the OutputStream.

The OutputStream class is the base class of all output streams in the Java IO API. Subclasses include the BufferedOutputStream and the FileOutputStream among others. OutputStream's are used for writing byte based data, one byte at a time. Here is an example:

```
OutputStream output = new FileOutputStream("c:\\data\\output-text.txt");

while(moreData) {

  int data = getMoreData();

  output.write(data);

}

output.close();
```

The write() method of an OutputStream takes an int which contains the byte value of the byte to write.

Subclasses of OutputStream may have alternative write() methods. For instance, the DataOutputStream allows you to write Java primitives like int, long, float, double, boolean etc. with its corresponding methods writeBoolean(), writeDouble() etc.

Following is the example to demonstrate InputStream and OutputStream:

```
import java.io.*;

public class fileStreamTest{

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

```
  try{
     byte bWrite [] = {11,21,3,40,5};
    OutputStream os = new FileOutputStream("C:/test.txt");
     for(int x=0; x < bWrite.length ; x++){
       os.write( bWrite[x] ); // writes the bytes
      }
     os.close();
    InputStream is = new FileInputStream("C:/test.txt");
     int size = is.available();
     for(int i=0; i< size; i++){
       System.out.print((char)is.read() + "  ");
      }
     is.close();
  }catch(IOException e){
    System.out.print("Exception");
   }
   }
}
```

The above code would create file test.txt and would write given numbers in binary format. Same would be output on the stdout screen.

11.8 File Navigation and I/O

There are several other classes that needs to be discussed to know the basics of File Navigation and I/O.

- File Class
- FileReader Class
- FileWriter Class

11.8.1 File Class

Java File class represents the files and directory pathnames in an abstract manner. This class is used for creation of files and directories, file searching, file deletion etc. The File object represents the actual file/ directory on the disk. Following syntax creates a new File instance from a parent abstract pathname and a child pathname string.

File(File parent, String child);

Following syntax creates a new File instance by converting the given pathname string into an abstract pathname.

File(String pathname)

Following syntax creates a new File instance from a parent pathname string and a child pathname string.

File(String parent, String child)

Following syntax creates a new File instance by converting the given file: URI into an abstract pathname.

File(URI uri)

Once we have File object in hand then there is a list of helper methods which can be used manipulate the files.

S.N.	**Methods with Description**
1	public String getName() Returns the name of the file or directory denoted by this abstract pathname.
2	public String getParent() Returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.
3	public File getParentFile() Returns the abstract pathname of this abstract pathname's parent, or null if this pathname does not name a parent directory.
4	public String getPath() Converts this abstract pathname into a pathname string.
5	public boolean isAbsolute() Tests whether this abstract pathname is absolute. Returns true if this abstract pathname is absolute, false otherwise
6	public String getAbsolutePath() Returns the absolute pathname string of this abstract pathname.
7	public boolean canRead() Tests whether the application can read the file denoted by this abstract pathname. Returns true if and only if the file specified by this abstract pathname exists and can be read by the application; false otherwise.
8	public boolean canWrite() Tests whether the application can modify to the file denoted by this abstract pathname. Returns true if and only if the file system actually contains a file denoted by this abstract pathname and the application is allowed to write to the file; false otherwise.
9	public boolean exists() Tests whether the file or directory denoted by this abstract pathname exists. Returns true if and only if the file or directory denoted by this abstract pathname exists; false otherwise
10	public boolean isDirectory() Tests whether the file denoted by this abstract pathname is a directory. Returns true if and only if the file denoted by this abstract pathname exists and is a directory; false otherwise.
11	public boolean isFile() Tests whether the file denoted by this abstract pathname is a normal file. A file is normal if it is not a directory and, in addition, satisfies other system-dependent criteria. Any non-directory file created by a Java application is guaranteed to be a normal file. Returns true if and only if the file denoted by this abstract pathname exists and is a normal file; false otherwise

12	public long lastModified() Returns the time that the file denoted by this abstract pathname was last modified. Returns a long value representing the time the file was last modified, measured in milliseconds since the epoch (00:00:00 GMT, January 1, 1970), or 0L if the file does not exist or if an I/O error occurs.
13	public long length() Returns the length of the file denoted by this abstract pathname. The return value is unspecified if this pathname denotes a directory.
14	public boolean createNewFile() throws IOException Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist. Returns true if the named file does not exist and was successfully created; false if the named file already exists.
15	public boolean delete() Deletes the file or directory denoted by this abstract pathname. If this pathname denotes a directory, then the directory must be empty in order to be deleted. Returns true if and only if the file or directory is successfully deleted; false otherwise.
16	public void deleteOnExit() Requests that the file or directory denoted by this abstract pathname be deleted when the virtual machine terminates.
17	public String[] list() Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.
18	public String[] list(FilenameFilter filter) Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
20	public File[] listFiles() Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname.
21	public File[] listFiles(FileFilter filter) Returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
22	public boolean mkdir() Creates the directory named by this abstract pathname. Returns true if and only if the directory was created; false otherwise
23	public boolean mkdirs() Creates the directory named by this abstract pathname, including any necessary but nonexistent parent directories. Returns true if and only if the directory was created, along with all necessary parent directories; false otherwise.
24	public boolean renameTo(File dest) Renames the file denoted by this abstract pathname. Returns true if and only if the renaming succeeded; false otherwise

25	public boolean setLastModified(long time) Sets the last-modified time of the file or directory named by this abstract pathname. Returns true if and only if the operation succeeded; false otherwise .
26	public boolean setReadOnly() Marks the file or directory named by this abstract pathname so that only read operations are allowed. Returns true if and only if the operation succeeded; false otherwise.
27	public static File createTempFile(String prefix, String suffix, File directory) throws IOException Creates a new empty file in the specified directory, using the given prefix and suffix strings to generate its name. Returns an abstract pathname denoting a newly-created empty file.
28	public static File createTempFile(String prefix, String suffix) throws IOException Creates an empty file in the default temporary-file directory, using the given prefix and suffix to generate its name. Invoking this method is equivalent to invoking createTempFile(prefix, suffix, null). Returns abstract pathname denoting a newly-created empty file.
29	public int compareTo(File pathname) Compares two abstract pathnames lexicographically. Returns zero if the argument is equal to this abstract pathname, a value less than zero if this abstract pathname is lexicographically less than the argument, or a value greater than zero if this abstract pathname is lexicographically greater than the argument.
30	public int compareTo(Object o) Compares this abstract pathname to another object. Returns returns zero if the argument is equal to this abstract pathname, a value less than zero if this abstract pathname is lexicographically less than the argument, or a value greater than zero if this abstract pathname is lexicographically greater than the argument.
31	public boolean equals(Object obj) Tests this abstract pathname for equality with the given object. Returns true if and only if the argument is not null and is an abstract pathname that denotes the same file or directory as this abstract pathname.
32	public String toString() Returns the pathname string of this abstract pathname. This is just the string returned by the getPath() method.

Following is the example to demonstrate File object:

```
import java.io.File;
class DirList {
  public static void main(String args[]) {
    String dirname = "/java";
    File f1 = new File(dirname);
    if (f1.isDirectory()) {
      System.out.println( "Directory of " + dirname);
      String s[] = f1.list();
      for (int i=0; i < s.length; i++) {
          File f = new File(dirname + "/" + s[i]);
          if (f.isDirectory()) {
            System.out.println(s[i] + " is a directory");
            } else {
            System.out.println(s[i] + " is a file");
            }
          }
      } else {
       System.out.println(dirname + " is not a directory");
    }
  }
}
```

This would produce following result:

```
Directory of /mysql
bin is a directory
lib is a directory
demo is a directory
test.txt is a file
README is a file
index.html is a file
include is a directory
```

11.8.2 FileReader Class

FileReader class inherits from the InputStreamReader class. FileReader is used for reading streams of characters. This class has several constructors to create required objects.

Here is a syntax which creates a new FileReader, given the File to read from.

FileReader(File file)

Following syntax creates a new FileReader, given the FileDescriptor to read from.

FileReader(FileDescriptor fd)

Following syntax creates a new FileReader, given the name of the file to read from.

FileReader(String fileName)

Once we have FileReader object in hand then there is a list of helper methods which can be used for the manipulation of the files.

S.N.	Methods with Description
1	public int read() throws IOException Reads a single character. Returns an int, which represents the character read.
2	public int read(char [] c, int offset, int len) Reads characters into an array. Returns the number of characters read.

Example:

Following is the example to demonstrate class:

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

```
public class FileRead{
  public static void main(String args[])throws IOException{
    File file = new File("Hello1.txt");
            // creates the file
    file.createNewFile();
            // creates a FileWriter Object
    FileWriter writer = new FileWriter(file);
    // Writes the content to the file
    writer.write("This\n is\n an\n example\n");
    writer.flush();
    writer.close();
     //Creates a FileReader Object
    FileReader fr = new FileReader(file);
     char [] a = new char[50];
    fr.read(a); // reads the content to the array
     for(char c : a)
       System.out.print(c); //prints the characters one by one
     fr.close();
   }
}
```

This would produce following result:

```
 This
 is
 an
 example
```

11.8.3 FileWriter Class

FileWriter class inherits from the OutputStreamWriter class. The class is used for writing streams of characters. This class has several constructors to create required objects.

Following syntax creates a FileWriter object given a File object.

FileWriter(File file)

Following syntax creates a FileWriter object given a File object.

FileWriter(File file, boolean append)

Following syntax creates a FileWriter object associated with a file descriptor.

FileWriter(FileDescriptor fd)

Following syntax creates a FileWriter object given a file name.

FileWriter(String fileName)

Following syntax creates a FileWriter object given a file name with a boolean indicating whether or not to append the data written.

FileWriter(String fileName, boolean append)

Once we have FileWriter object in hand then there is a list of helper methods which can be used manipulate the files.

S.N.	Methods with Description
1	public void write(int c) throws IOException Writes a single character.
2	public void write(char [] c, int offset, int len) Writes a portion of an array of characters starting from offset and with a length of len.
3	public void write(String s, int offset, int len) Write a portion of a String starting from offset and with a length of len.

Following is the example to demonstrate class:

```
import java.io.*;
public class FileRead{
  public static void main(String args[])throws IOException{
    File file = new File("Hello1.txt");
            // creates the file
    file.createNewFile();
            // creates a FileWriter Object
    FileWriter writer = new FileWriter(file);
    // Writes the content to the file
    writer.write("This\n is\n an\n example\n");
    writer.flush();
    writer.close();
     //Creates a FileReader Object
    FileReader fr = new FileReader(file);
     char [] a = new char[50];
    fr.read(a); // reads the content to the array
     for(char c : a)
       System.out.print(c); //prints the characters one by one
     fr.close();
   }
}
```

This would produce following result:

```
 This
 is
 an
 example
```

11.9 Summary

- •A stream can be defined as a sequence of data. The InputStream is used to read data from a source and the OutputStream is used for writing data to a destination.
- •The System class provides facilities such as the standard input, output and error streams.
- •An I/O Stream represents an input source or an output destination.
- •Programs use byte streams to perform input and output of 8-bit bytes. All byte stream classes are descended from InputStream and OutputStream.

•The Java platform stores character values using Unicode conventions. Character stream I/O automatically translates this internal format to and from the local character set. In Western locales, the local character set is usually an 8-bit superset of ASCII.

•FileInputStream is used for reading data from the files. Objects can be created using the keyword new and there are several types of constructors available.

•FileOutputStream is used to create a file and write data into it.

11.10 Self-Assessment Questions

1. Define I/O streams.
2. What is the difference between InputStream and OutputStream?
3. Describe the facilities provided by the System class?
4. Write a short note on byte stream and character stream.
5. Explain the file navigation and I/O.

11.11 References

1. Programming with Java A Primer, E.Balaguruswamy Tata McGraw Hill Companies
2. Java Programming John P. Flynt Thomson 2nd
3. Java Programming Language Ken Arnold Pearson
4. The complete reference JAVA2, Herbert schildt. TMH
5. Big Java, Cay Horstmann 2nd edition, Wiley India Edition
6. Core Java, Dietel and Dietel
7. Java - Balaguruswamy
8. Java server programming, Ivan Bayross SPD

Unit - 12
Graphical User Interface

Structure of the Unit

12.0 Objective

This chapter provides a general overview of

- Graphical User Interface
- Class hierarchy for Panel and Frame Class
- The Window Class
- Working with Window class
- How to add functionality to GUI
- Some improvements in GUI layout

12.1 Introduction

Computer users today expect to interact with their computers using a graphical user interface (GUI). Java can be used to write GUI programs ranging from simple applets which run on a Web page to sophisticated stand-alone applications. GUI programs differ from traditional "straight-through" programs that GUI programs are event-driven. That is, user actions such as clicking on a button or pressing a key on the keyboard generate events, and the program must respond to these events as they occur. Objects are everywhere in GUI programming. Events are objects. Colors and fonts are objects. GUI components such as buttons and menus are objects. Events are handled by instance methods contained in objects. In Java, GUI programming is object-oriented programming.

12.2 Basics of GUI Applications

There are two basic types of GUI program in Java: stand-alone applications and applets. An applet is a program that runs in a rectangular area on a Web page. Applets are generally small programs, meant to do fairly simple things, although there is nothing to stop them from being very complex. Applets were responsible for a lot of the initial excitement about Java when it was introduced, since they could do things that could not otherwise be done on Web pages. However, there are now easier ways to do many of the more basic things that can be done with applets, and they are no longer the main focus of interest in Java. Nevertheless, there are still some things that can be done best with applets, and they are still somewhat common on the Web. We will look at applets in the next chapter.

A stand-alone application is a program that runs on its own, without depending on a Web browser. Any class that has a main() routine defines a stand-alone application; running the program just means executing this main() routine. However, the programs that you've seen up till now have been "command-line" programs, where the user and computer interact by typing things back and forth to each other. A GUI program offers a much richer type of user interface, where the user uses a mouse and keyboard to interact with GUI components such as windows, menus, buttons, check boxes, text input boxes, scroll bars, and so on. The main routine of a GUI program creates one or more such components and displays them on the computer screen. Once a GUI component has been created, it follows its own programming---programming that tells it how to draw itself on the screen and how to respond to events such as being clicked on by the user.

There are two sets of Java APIs for graphics programming: AWT (Abstract Window Toolkit) and Swing.

1. AWT API (Application Program Interface) was introduced in JDK 1.0.

2. Swing API, a much more comprehensive set of graphics libraries that enhances the AWT, was introduced as part of Java Foundation Classes (JFC) after the release of JDK 1.1.

Java Graphics APIs - AWT and Swing - provide a huge set of reusable GUI components, such as button, text field, label, choice, panel and frame for building GUI applications. You can simply reuse these classes rather than re-invent the wheels.

AWT consists of 12 packages. Only 2 packages java.awt and java.awt.event are commonly used.

1. The java.awt package contains the core AWT graphics classes:
   - o GUI Component classes (such as Button, TextField, and Label),
   - o GUI Container classes (such as Frame, Panel, Dialog and ScrollPane),
   - o Layout managers (such as FlowLayout, BorderLayout and GridLayout),
   - o Custom graphics classes (such as Graphics, Color and Font).
2. The java.awt.event package supports event handling:
   - o Event classes (such as ActionEvent, MouseEvent, KeyEvent and WindowEvent),
   - o Event Listener Interfaces (such as ActionListener, MouseListener, KeyListener and WindowListener),
   - o Event Listener Adapter classes (such as MouseAdapter, KeyAdapter, and WindowAdapter).

AWT provides a platform-independent and device independent interface to develop graphic programs that runs on all platforms e.g. Windows, Mac, Unix, etc.

In a Java GUI program, each GUI component in the interface is represented by an object in the program. One of the most fundamental types of component is the Window. The Window class is a subclass of the Container class that provides a common set of methods for implementing windows. The Window class has two subclasses, Frame and Dialog that are used to create Window objects. The Frame class is used to create a main application window, and the Dialog class is used to implement dialog boxes.

12. 3 Window 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. Much of the functionality of these windows is derived from their parent classes. Thus, a description of the class hierarchies relating to these two classes is fundamental to their understanding. Figure 12.1 shows the class hierarchy for Panel and Frame.

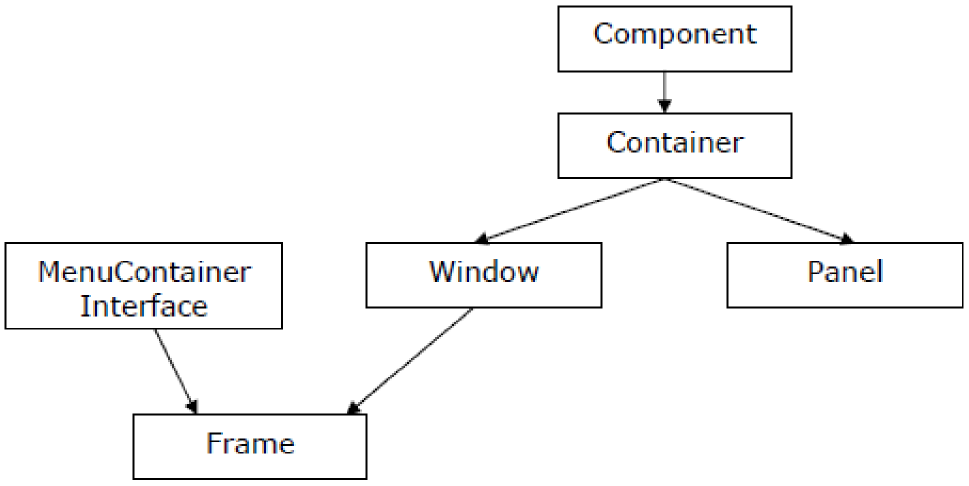


Fig 12.1 Class hierarchy for Panel and Frame

12.3.1 Component

At the top of the AWT hierarchy is the Component class. Component is an abstract class that encapsulates all of the attributes of a visual component. All user interface elements that are displayed on the screen and that interact with the user are subclasses of Component. It defines over a hundred public methods that are responsible for managing events, such as mouse and keyboard input, positioning and sizing the window, and repainting. A Component object is responsible for remembering the current foreground and background colors and the currently selected text font.

12.3.2 Container

The Container class is a subclass of Component. It has additional methods that allow other Component objects to be nested within it. Other Container objects can be stored inside of a Container (since they are themselves instances of Component). This makes for a multileveled containment system. A container is responsible for laying out (positioning) any components that it contains. It does this through the use of various layout managers.

12.3 3 Panel

The Panel class is a concrete subclass of Container. It doesn't add any new methods; it simply implements Container. A Panel may be thought of as a recursively nestable, concrete screen component. Panel is the super-class for Applet. When screen output is directed to an applet, it is drawn on the surface of a Panel object. In essence, a Panel is a window that does not contain a title bar, menu bar, or border. This is why we don't see these items when an applet is run inside a browser. When we run an applet using an applet viewer, the applet viewer provides the title and border. Other components can be added to a Panel object by its add ( ) method (inherited from Container). Once these components have been added, we can position and resize them manually using the setLocation( ), setSize( ), or setBounds( ) methods defined by Component.

12.3.4 Window

The Window class creates a top-level window. A top level window is not contained within any other object; it sits directly on the desktop. Generally, we won't create Window objects directly. Instead, we will use a subclass of Window called Frame.

12.3.5 Frame

Frame encapsulates what is commonly thought of as a "window." It is a subclass of Window and has a title bar, menu bar, borders, and resizing corners. If we create a Frame object from within an applet, it will contain a warning message, such as "Java Applet Window," to the user that an applet window has been created. This message warns users that the window they see was started by an applet and not by software running on their computer. When a Frame window is created by a program rather than an applet, a normal window is created.

12.3.6 Canvas

Canvas encapsulates a blank window upon which we can draw.

Dimension

This class encapsulates the 'width' and 'height' of a component (in integer precision) in a single object. The class is associated with certain properties of components. Several methods defined by the Component class and the Layout Manager Interface return a Dimension object.

Normally the values of width and height are non negative integers. The constructors that allow us to create a dimension do not prevent us from setting a negative value for these properties. If the value of width or height is negative, the behavior of some methods defined by other objects is undefined.

Fields of Dimension

int height	The height dimension; negative values can be used.
int width	The width dimension; negative values can be used.

Constructors

Dimension()

It creates an instance of Dimension with a width of zero and a height of zero.

Dimension(Dimension d)

It creates an instance of Dimension whose width and height are the same as for the specified dimension.

Dimension(int width, int height)

It constructs a Dimension and initializes it to the specified width and specified height.

Methods

boolean equals(Object obj)

It checks whether two dimension objects have equal values.

double getHeight()

It returns the height of this dimension in double precision.

Dimension getSize()

It gets the size of this Dimension object.

double getWidth()

It returns the width of this dimension in double precision.

void setSize(Dimension d)

It sets the size of this Dimension object to the specified size.

void setSize(double width, double height)

It sets the size of this Dimension object to the specified width and height in double precision.

void setSize(int width, int height)

It sets the size of this Dimension object to the specified width and height.

12.4 Developing a JAVA GUI using Frame

Other than the applet, the type of window we will most often create is derived from Frame. We will use it to create child windows within applets, and top-level or child windows for applications. It creates a standard -style window. Following are two of Frame's constructors:

Frame( )

is used to creates a standard window that does not contain a title.

Frame(String title)

creates a window with the title specified by title.

Note that we cannot specify the dimensions of the window. Instead, we must set the size of the window after it has been created.

Setting the Windows Dimensions

The setSize( ) method is used to set the dimensions of the window. Its signature is shown below:

void setSize(int newWidth, int newHeight)

void setSize(Dimension newSize)

The new size of the window is specified by 'newWidth' and 'newHeight', or by the 'width' and 'height' fields of the Dimension object passed in 'newSize'. The dimensions are specified in terms of pixels. The getSize( ) method is used to obtain the current size of a window. Its signature is:

Dimension getSize( )

This method returns the current size of the window contained within the 'width' and 'height' fields of a Dimension object.

Hiding and showing a Window

After a frame window has been created, it will not be visible until we call setVisible( ). Its signature is:

void set Visible(boolean visibleFlag)

The component is visible if the argument to this method is true. Otherwise, it is hidden.

Setting a Windows Title

We can change the title in a frame window using setTitle( ), which has this general form:

void setTitle(String newTitle)

Here, 'newTitle' is the new title for the window.

Closing a Frame Window

When using a frame window, our program must remove that window from the screen when it is closed, by calling set Visible(false). To intercept a window close event, we must implement the windowClosing( ) method of the Window Listener interface. Inside windowClosing( ), we must remove the window from the screen.

12.5 Adding Functionality to GUI

Creating Frame

This program shows you how to create a frame in java AWT package. The frame in java works like the main window where your components (controls) are added to develop an application. In the Java AWT, top level windows are represented by the Frame class. Java supports the look and feel and decoration for the frame. For creating java standalone application you must provide GUI to the user.

The most common method of creating a frame is by using single argument constructor of the Frame class that contains the single string argument which is the title of the window or frame. Then you can add user interface by constructing and adding different components to the container one by one.

Label.CENTER. The frame initially invisible, so after creating the frame it need to visualize the frame by setVisible(true) method.

add(lbl)

This method has been used to add the label to the frame. Method add() adds a component to it's container.

setSize (width, height):

This is the method of the Frame class that sets the size of the frame or window. This method takes two arguments width (int), height (int).

setVisible(boolean):

This is also a method of the Frame class sets the visibility of the frame. The frame will be invisible if you pass the boolean value false otherwise frame will be visible.

Here is the code of the program:

```
import java.awt.*;

public class AwtFrame{

public static void main(String[] args){

Frame frm = new Frame("Java AWT Frame");

Label lbl = new Label("Welcome",Label.CENTER);

frm.add(lbl);
frm.setSize(400,400);
frm.setVisible(true);
  }
}
```

Example: Create Frame Window Example

```
/*This java example shows how to create frame window using Java AWT.*/
  import java.awt.Frame;
```

Here "awt" stands for "Abstract Window Toolkit". Frame is a class inside the package awt.

```
/* To create a stand alone window, class should be extended from
Frame and not from Applet class. */
public class CreateFrameWindowExample extends Frame{
CreateFrameWindowExample(String title){
//call the superclass constructor
super();
//set window title using setTitle method
this.setTitle(title);
/* Newly created window will not be displayed until we call
setVisible(true).*/
this.setVisible(true);
}
public static void main(String args[]){
CreateFrameWindowExample window = new CreateFrameWindowExample("Create Window Example");
/* In order to close the window, we will have to handle the events
and call setVisible(false) method. This Example program does not handle events, so you will have to
terminate the program (by pressing ctrl+C) in a terminal window
to close the frame window. */
}
}
```

Output is shown in figure 12.2.

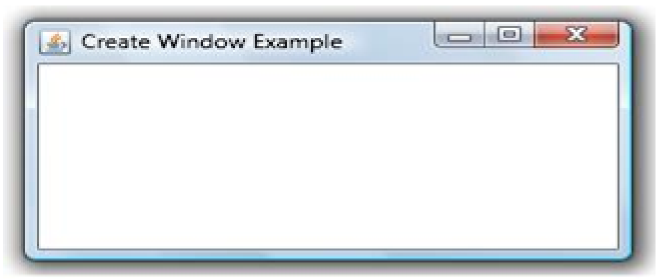


Figure 12.2 Output of create Frame window

12.6 Improving GUI layout

A perfect GUI has various controls embedded on the interface. The controls are as follows-

- Label
- Button
- Checkbox

- Choice
- List
- Scroll bar
- Textfield
- Textarea

All of the above mentioned controls are positioned by some layout manger. A layout manager is an instance of any class that implements the interface LayoutManager. This layout manager automatically arranges the controls within a window by using some type of algorithm. Java has several LayoutManager classes-

1. FlowLayout
2. BorderLayout
3. GridLayout
4. CardLayout

The details of these layout managers are covered in Unit-15.

12.7 Summary

We can write and run GUI based application and applets by using Frame class and Applet class. Frame is a subclass of Window class which is again a subclass of Container class. Component class is the top most class in hierarchy which is the super class of Container class. Various AWT controls can be created on GUI and they can be positioned according to existing layout manager classes.

12.8 Self-Assessment Questions

1. Explain the following classes with their positions in the hierarchy-
    a. Frame
    b. Window
    c. Panel
2. Write a java application program to create a GUI. Use appropriate AWT controls.
3. Explain any four methods of class Frame.

12.9 References

1. Programming with Java A Primer, E.Balaguruswamy Tata McGraw Hill Companies
2. Java Programming John P. Flynt Thomson 2nd
3. Java Programming Language Ken Arnold Pearson
4. The complete reference JAVA2, Herbert schildt. TMH
5. Big Java, Cay Horstmann 2nd edition, Wiley India Edition
6. Core Java, Dietel and Dietel
7. Java - Balaguruswamy
8. Java server programming, Ivan Bayross SPD

Unit - 13
Applets

Structure of the Unit

13.0 Objective

This chapter provides a general overview of

- Applet
- Difference between Java Applet and Application
- Applet life cycle
- Working of the Applet
- How to embed Applet in HTML file
- Brief overview of java.Applet package

13.1 Introduction

Applets are the powerful tools because it covers half of the java language picture. Java applets are the best way of creating the programs in java. A Java applet is a small dynamic Java program that can be transferred via the Internet and run by a Java-compatible Web browser. Applet can be embedded into HTML pages. Java applets run on the java enables web browsers such as Mozilla and Internet Explorer. Applet is designed to run remotely on the client browser, so there are some restrictions on it. Applet can't access system resources on the local computer. Applets are used to make the web site more dynamic and entertaining.

13.2 Java Application Vs Java Applets

All Java programs can be classified as Applications and Applets. The striking differences are that applications contain main() method where as applets do not. One more is, applications can be executed at DOS prompt and applets in a browser. We can say, an applet is an Internet application. Noticeable differences are:

- Applets are the small programs while applications are larger programs.
- Applets don't have the main method while in an application execution starts with the main method.
- Applets can run in our browser's window or in an appletviewer. To run the applet in an appletviewer will be an advantage for debugging. An application is a Java program that runs all by itself.
- Applets are designed just for handling the client site problems while the java applications are designed to work with the client as well as server.
- Applications are designed to exist in a secure area while the applets are typically used.
- Applications and applets have much of the similarity such as both have most of the same features and share the same resources.
- Applets are created by extending the java.applet.Applet class while the java applications start execution from the main method.

These Differences can be summarized as:

Feature	**Application**	**Applet**
main() method	PresentNot	present
Execution	Requires JRE	Requires a browser like Chrome
Nature	Called as stand-alone application	Requires some third party tool help like a browser
Restrictions	Can access complete system	cannot access anything on the system except browser
Security	Does not require any security	Requires highest security for the system as they are untrusted

Table 13.1: Differences between Application and Applet

Less number of java programmers have the hands on experience on java applications. This is not the deficiency of java applications but the global utilization of internet. It doesn't mean that the java applications don't have the place. Both (Applets and the java applications) have the same importance at their own places. Both Applications and Applets are the platform independent as well as byte oriented. The key feature is that while they have so many differences but both can perform the same purpose.

13.3 Simple Applets

Let's begin with the simple applet -Hello World Applet Example. This java example shows how to create Hello World Java Applet.

import java.applet.Applet;

```
 import java.awt.*;
 /* <applet code="HelloWorld" width=300 height=400>
</applet>  */
 public class HelloWorld extends Applet {
       public void paint(Graphics g) {
      g.setFont(new Font("Helvetica", Font.PLAIN, 8));
      g.drawString("Hello world!", 50, 25);
       }//paint
   }//HelloWorld
```

Java supplies a huge library of pre-written "code," ready for you to use in your programs, these codes are organized into classes which are grouped into packages. One way to use this code is to import it; you can import a single class, or all the classes in a package. To create an applet, you must import the Applet class, This class is in the java.applet package. The Applet class contains code that works with a browser to create a display window. There is a difference in applet and Applet, applet is the package whereas Applet is the name of class.

In the program the first line of the code is

```
   import java.applet.Applet;
```

Here import is a keyword, java.applet is the name of the package. A dot (.) separates the package from the class, Applet is the name of the class. There is a semicolon (;) at the end.

The next line in the code is

```
 import java.awt.*;
```

Here "awt" stands for "Abstract Window Toolkit". The java.awt package includes classes for:

- Drawing lines and shapes
- Drawing letters
- Setting colors
- Choosing fonts

Since you may want to use many classes from the java.awt package, simply import them all. here The asterisk, or star (*), means "all classes". The import directives can go in any order, but must be the first lines in your program.

```
public class HelloWorld extends Applet{
   …  }
```

Here HelloWorld is the name of your class and Class names should always be capitalized.

" extends Applet says that our HelloWorld is a kind of Applet, but with added capabilities

o Java's Applet just makes an empty window

o We are going to draw in that window

The only way to make an applet is to extend Applet. Our applet is going to have a method to paint some text on the screen. This method must be named paint. paint needs to be told where on the screen it can draw, This will be the only parameter it needs, paint doesn't return any result.

public void paint(Graphics g) { … }

- public says that anyone can use this method
- void says that it does not return a result

A Graphics (short for "Graphics context") is an object that holds information about a painting

- It remembers what color you are using
- It remembers what font you are using
- You can "paint" on it (but it doesn't remember what you have painted)

To write a text, we tell our Graphics g what font we want:

g.setFont(new Font("Helvetica", Font.PLAIN, 8));

- g will remember this font and use it for everything until we tell it some different font

Java uses an (x, y) coordinate system, (0, 0) is the top left corner, (50, 0) is 50 pixels to the right of (0, 0), (0, 20) is 20 pixels down from (0, 0). To draw a string " Hello world!" at location (50,25) we use

```
g.drawString("Hello world!", 50, 25);
```

Hence the complete code for the HelloWorld applet has to be a public class called "HelloWorld" that extends Applet and is in a Java file called: HelloWorld.java.

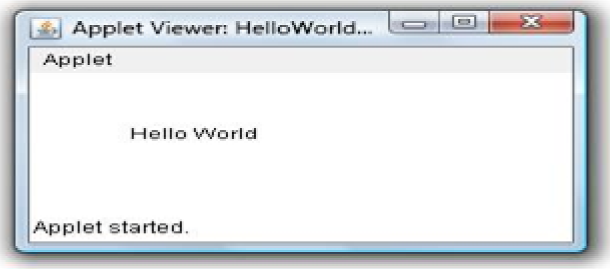
In general, you can quickly iterate through applet development by using following three steps:

1. Edit a java source file.
2. Compile your program.
3. Execute the applet viewer, specifying the name of your applet's source file. The applet viewer will encounter the APPLET tag within the comment and execute your applet.

Hence the java source file of HelloWorld applet is compile and run as following:

```
> javac HelloWorld.java
> appletviewer Helloworld.java
```

The window produced by HelloWorld, as displayed by the applet viewer, is shown in Fig 13.1.



```
Hello World
Applet started.
```

Figure 13.1 : Output-Hello world Applet

Let us take an another example of Applet

```
import java.applet.Applet;
import java.awt.*;
/*< applet code="Drawing" height=200 width=300>
</applet> */
public class Drawing extends Applet {
   public void paint(Graphics g) {
      g.setColor(Color.BLUE);
       g.fillRect(20, 20, 50, 30);
      g.setColor(Color.RED);
       g.fillRect(50, 30, 50, 30);
    }
}
```

When we compile and run the output is as shown figure 13.2. It display two rectangle Blue and red in window.

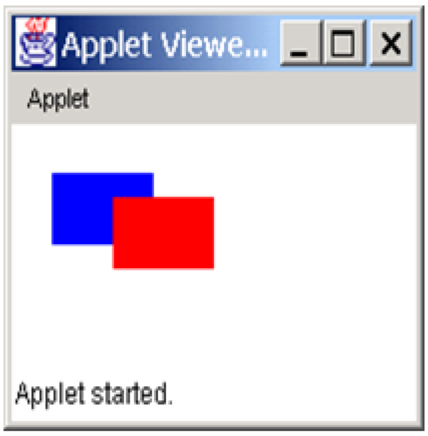


Figure 13.2: Drawing Applet output

13.4 Applet Life Cycle

As seen in our first Applet example, the Applet class inherits the paint method from the Container class. There are four additional methods that you will often need to implement in your applets: init, start, stop, and destroy. Each of these corresponds to a major event in the life of an applet. Let us see what each of them does.

- init()

This method is like a constructor. It is called by the browser (or applet viewer) to inform the applet that it has been loaded into the system. Hence, it is called before any of the other three methods are called, and is good for any initialization that needs to be done. The Applet class provides a default implementation of this method that does nothing.

- start()

This method is called after the init method. It is also called after the page has been maximized or revisited. The purpose of the method is to inform the applet that it should start executing.

When should you use init and when should you use start? Since the start method can be called more than once and the init method is called only once, any code that you want to execute only once should be put in init.

On the other hand, you should place in start the code that you want to execute every time a user visits the page containing your applet. For instance, in cases where the applet contains an animation, you would want to start the animation every time the applet becomes visible.

As with init, the default implementation of the start method does nothing.

- stop()

This method is called when the user changes pages, when the page is minimized, and just before the applet is destroyed. It is called by the browser (or applet viewer) to inform the applet to stop executing (think of the aforementioned animation example).

If you are explicitly starting an activity with the start method, you will probably want to stop it with the stop method. For example, you may want to stop and start any resource-intensive activity as the user leaves and returns to the page. The default implementation of stop does nothing.

- destroy()

This method is called by the browser or the applet viewer to inform the applet that it is being destroyed and that it should release any resources that it has allocated. The default implementation of destroy does nothing.

To get a clearer picture of what an applet goes through in its lifetime, let us look at an example:

```
/*
 <applet code="LifeCycle.class" width=300 height=300></applet>
*/
import java.applet.Applet;
import java.awt.*;
public class LifeCycle extends Applet
{
    String output = "";
    String event;
    public void init()
    {
        event = "Initializing...";
           printOutput();
    }
    public void start()
```

```
     {
        event = "Starting...";
              printOutput();
     }
    public void stop()
     {
        event = "Stopping...";
              printOutput();
     }
    public void destroy()
     {
        event = "Destroying...";
              printOutput();
     }
    private void printOutput()
     {
        System.out.println(event);
              output += event;
              repaint();
     }
   public void paint(Graphics g)
     {
        g.drawString(output, 10, 10);
     }
}
```

Compile and run the applet:

```
> javac LifeCycle.java
> appletviewer LifeCycle.java
```

and watch what happens as you minimize and maximize the appletviewer window. Also, watch the event messages as they are printed to the terminal. You should end up with something that looks like:

```
Initializing...
Starting...
Stopping...
Starting...
Stopping...
Starting...
Stopping...
Destroying...
```

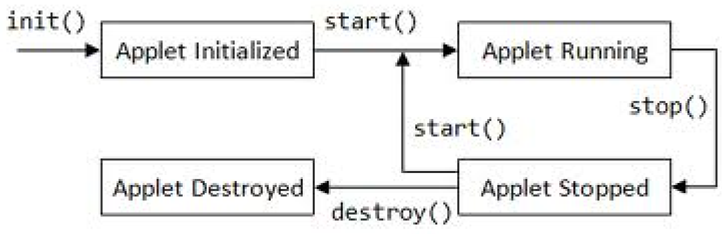


Figure 13.3: Flow in Applet Life Cycle

13.5 Advantages of Applets

1. Deployment of applets is easy in a Web browser and does not require any installation procedure in realtime programming.
2. Writing and displaying (just opening in a browser) graphics and animations is easier than applications.
3. In GUI development, constructor, size of frame, window closing code etc. are not required (but are required in applications).

13.6 Restrictions of Applets

1. Applets require a compiler to compile and also a browser to execute.
2. In realtime environment, the bytecode of applet is to be downloaded from the server to the client machine.
3. Applets are treated as untrusted and for this reason they are not allowed, as a security measure, to access any system resources like file system etc. available on the client system.
4. Code can be developed to communicate between applets using AppletContext.

What Applet can't do - Security Limitations

Applets are treated as untrusted because they are developed by somebody and placed on some unknown Web server. When downloaded, they may harm the system resources or steal passwords and valuable information. As applets are untrusted, the browsers come with many security restrictions. Security policies are browser dependent. Browser does not allow the applet to access any of the system resources.

- Applets are not permitted to use any system resources like file system as they are untrusted and can inject virus into the system.
- Applets cannot read from or write to hard disk files.
- Applet methods cannot be native.
- Applets should not attempt to create socket connections.

13.7 The HTML Applet Tag

When you put an applet on your page you will need to save the applet on your server as well as the HTML page the applet is embedded in. When the page is loaded by a visitor the applet will be loaded and inserted on the page where you embedded it.

Applets have the file extension "class". An example would be "Helloworld.class". Some applets consist of more than just one class file, and often other files need to be present for the applet to run (such as JPG or GIF images used by the applet). Make sure to check the documentation for the applet to see if you have all files for it to run.

Before embedding an applet on your page you need to upload the required files to your server.
Next is a short example showing how simple it is to embed an applet on the html page.

```
<APPLET CODE="ClassName.class" HEIGHT=h WIDTH=w>
 Alternate text
 </APPLET>
```

- Here h and w are the HEIGHT and WIDTH of the box in which the applet outputs its response. The alternate text appears when a browser cannot handle APPLETs. The code in "ClassName.class" above, is the result of compiling a file called "ClassName.java" that contains a public class called ClassName which extends an Applet.

Here is a suitable piece of HTML to test a simple HelloWorld class, Put this in a file called: test.HelloWorld.html

- <head>
- <APPLET CODE="HelloWorld.class" HEIGHT=150 WIDTH=150>
- You can not see this brilliant Java Applet.
- </APPLET>
- </body>

Two HTML tags are relevant according to applets: <Applet> and <Param>.

- The <Applet> tag embeds the applet in your HTML page.
- The <Param> tag is used to enter parameters for the applet.

The Table 13.2 shows the attributes which can be set for the <Applet> tag.

Attribute	**Explanation**	**Example**

Code
Width=n
Height=n
	Name of class file
	n=Width of applet
	n=Height of applet
		Code="myapplet.class"
		Width=200
		Height=100

Codebase	Library where the applet is stored. If the applet is in same directory as your page this can be omitted.	Codebase="applets/"
Alt="Text"	Text that will be shown in browsers where the ability to show applets has been turned off.	alt="Menu Applet"
Name=Name	Assigning a name to an applet can be used when applets should communicate with each other.	Name="starter"
Align= Left Right Top Texttop Middle Absmiddle Baseline Bottom Absbottom	Justifies the applet according to the text and images surrounding it.	Align=Right

Table 13.2: Attributes of Applet tag

<Param> tag has the general syntax:

<Param Name=NameOfParameter Value="ValueOfParameter">

Each applet has different parameters that should be set.

Typical parameters for an applet would be:

- color used by the applet
- font and font size to be used on text in the applet
- name of an image file to be inserted in the applet

To see which parameters should be set by a specific applet, you will need to read the documentation for that applet. It will list names and possible values for parameters.

Applet Tag Example

The APPLET tag is also a container tag. Its contents can include one or more <PARAM> tags which are applet specific. Each PARAM tag represents one name, value pair that is made available to the applet while it is running. For example, if the applet was a ticker tape banner, the programmer who developed the banner might have the applet look for a PARAM tag with the name MESSAGE. The value assigned to message would be the text the ticker tap would display:

<PARAM NAME="MESSAGE" VALUE="Message of the day">

If there is text included between the APPLET start and end tag, this text is displayed on Web browsers that cannot run Java applets. Here is an example of a Java applet that acts as a ticker tape or displays the message "Java applet here" if the browser cannot run applets:

```
<APPLET CODE="http://server.edu/java/ticker.class" HEIGHT="200" WIDTH="400">

<EM>Java applet here</EM>

<PARAM NAME="MESSAGE" VALUE="The news of the day">

</APPLET>
```

Here is an example of a simple APPLET tag:

```
 <applet code="MyApplet.class" width=100 height=140></applet>
```

This tells the viewer or browser to load the applet whose compiled code is in MyApplet.class (in the same directory as the current HTML document), and to set the initial size of the applet to 100 pixels wide and 140 pixels high.

```
 <applet codebase="http://java.sun.com/applets/NervousText/1.1"

        code="NervousText.class" width=400 height=75>

 <param name="text" value="Welcome to HotJava!">

 <hr>

 If you were using a Java-enabled browser such as HotJava,

 you would see dancing text instead of this paragraph.

 </hr>

 </applet>
```

This tells the viewer or browser to load the applet whose compiled code is at the URL http://java.sun.com/applets/NervousText/1.1/NervousText.class, to set the initial size of the applet to 400x75 pixels. The viewer/browser must also set the applet's "text" attribute (which customizes the text this applet displays) to be "Welcome to HotJava!" If the page is viewed by a browser that can't execute Java applets, then the browser will ignore the APPLET and PARAM tags, displaying only the HTML between the <param> and </applet> tags (the alternate HTML).

13.8 The java.applet Package

Contents:

java.applet.Applet (JDK 1.0)

java.applet.AppletContext (JDK 1.0)

java.applet.AppletStub (JDK 1.0)

java.applet.AudioClip (JDK 1.0)

An applet is a small, embeddable Java program. The java.applet package is a small one. It contains the Applet class, which is the superclass of all applets, and three related interfaces. Figure 13.1 shows the class hierarchy of this package.

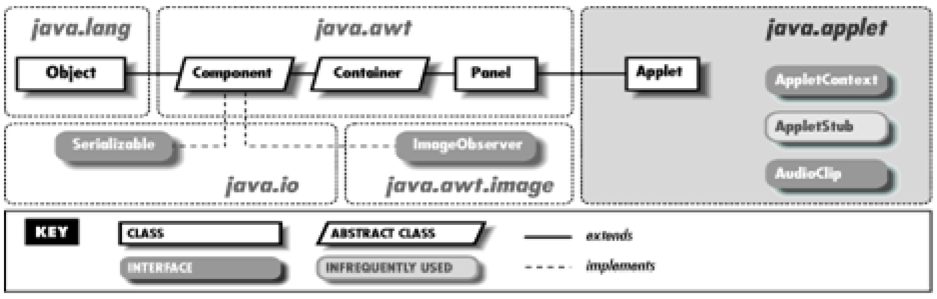


Figure 13.4: The java.applet package

13.8.1 java.applet.Applet

This class implements an applet. To create your own applet, you should create a subclass of this class and override some or all of the following methods. Note that you never need to call these methods--they are called when appropriate by a Web browser or other applet viewer.

init() should perform any initialization for the applet; it is called when the applet first starts. destroy() should free up any resources that the applet is holding; it is called when the applet is about to be permanently stopped. start() is called to make the applet start doing whatever it is that it does. Often, it starts a thread to perform an animation or similar task. stop() should temporarily stop the applet from executing. It is called when the applet temporarily becomes hidden or non-visible.

getAppletInfo() should return text suitable for display in an About dialog posted by the Web browser or applet viewer. getParameterInfo() should return an arbitrary-length array of three-element arrays of strings where each element describes one of the parameters that this applet understands. The three elements of each parameter description are strings that specify, respectively, the parameter's name, type, and description.

In addition to these methods, an applet also typically overrides several of the methods of java.awt.Component, notably the paint() method to draw the applet on the screen. There are also several Applet methods that you do not override but may call from applet code: showStatus() displays text in the Web browser or applet viewer's status line. getImage() and getAudioClip() read image (GIF and JPEG formats) and audio files (AU format) over the network and return corresponding Java objects. getParameter() looks up the value of a parameter specified with a <PARAM> tag within an <APPLET>...</APPLET> pair. getCodeBase() returns the base URL from which the applet's code was loaded, and getDocumentBase() returns the base URL from which the HTML document containing the applet was loaded. getAppletContext() returns an AppletContext object, which also has useful methods.

```
public class Applet extends Panel {
   // Default Constructor: public Applet()
   // Public Instance Methods
          public void destroy();
         public AppletContext getAppletContext();
         public String getAppletInfo();
         public AudioClip getAudioClip(URL url);
         public AudioClip getAudioClip(URL url, String name);
          public URL getCodeBase();
         public URL getDocumentBase();
         public Image getImage(URL url);
         public Image getImage(URL url, String name);
         public Locale getLocale();  // Overrides Component
         public String getParameter(String name);
         public String[][] getParameterInfo();
          public void init();
          public boolean isActive();
          public void play(URL url);
         public void play(URL url, String name);
         public void resize(int width, int height);  // Overrides Component
         public void resize(Dimension d);  // Overrides Component
         public final void setStub(AppletStub stub);
         public void showStatus(String msg);
          public void start();
          public void stop();
}
```

Hierarchy:

Object->Component(ImageObserver, MenuContainer, Serializable)->Container->Panel->Applet

Returned By:

AppletContext.getApplet()

13.8.2 java.applet.AppletContext

This interface defines the methods that allow an applet to interact with the context in which it runs (which is usually a Web browser or an applet viewer). The object that implements the AppletContext interface is returned by Applet.getAppletContext(). You can use it to take advantage of a Web browser's cache, or to display a message to the user in the Web browser's or applet viewer's message area.

The getAudioClip() and getImage() methods may make use of a Web browser's caching mechanism. showDocument() and showStatus() give an applet a small measure of control over the appearance of the browser or applet viewer. The getApplet() and getApplets() methods allow an applet to find out what other applets are running at the same time.

```
public abstract interface AppletContext {
   // Public Instance Methods
         public abstract Applet getApplet(String name);
         public abstract Enumeration getApplets();
         public abstract AudioClip getAudioClip(URL url);
         public abstract Image getImage(URL url);
         public abstract void showDocument(URL url);
         public abstract void showDocument(URL url, String target);
         public abstract void showStatus(String status);
}
```

Returned By:

Applet.getAppletContext(), AppletStub.getAppletContext()

13.8.3 java.applet.AppletStub

This is an internal interface used when implementing an applet viewer.

```
public abstract interface AppletStub {
   // Public Instance Methods
         public abstract void appletResize(int width, int height);
         public abstract AppletContext getAppletContext();
         public abstract URL getCodeBase();
         public abstract URL getDocumentBase();
         public abstract String getParameter(String name);
         public abstract boolean isActive();
}
```

Passed To:

Applet.setStub()

13.8.4 java.applet.AudioClip

This interface describes the essential methods that an audio clip must have. AppletContext.getAudioClip() and Applet.getAudioClip() both return an object that implements this interface. The current JDK implementations of this interface only work with sounds encoded in AU format. The AudioClip interface is in the java.applet package only because there is not a better place for it.

```
public abstract interface AudioClip {
   // Public Instance Methods
         public abstract void loop();
         public abstract void play();
```

```
    public abstract void stop();
}
```

Returned By:

Applet.getAudioClip(), AppletContext.getAudioClip()

13.9 Summary

All Java programs can be classified as Applications and Applets. Application is a Java class that has a main() method. An applet is a Java class which extends java.applet.Applet. Generally, application is a stand-alone program, normally launched from the command line, and which has unrestricted access to the host system. An applet is a program which is run in the context of an applet viewer or web browser, and which has strictly limited access to the host system. Two HTML tags are relevant according to applets: <Applet> and <Param>.The <Applet> tag embeds the applet in your HTML page. The <Param> tag is used to enter parameters for the applet. The java.applet package is a small one. It contains the Applet class, which is the superclass of all applets, and three related interfaces AppletContext, AppletStub, and AudioClip.

13.10 Self-Assessment Questions

1. What is the difference between Applet and Application?
2. What is the difference between applet and Applet?
3. Define Applet and write a Applet to print "My First Applet in Java"?
4. Explain the Applet tag of HTML?
5. What are the contents of java.applet package?

13.11 References

1. Programming with Java A Primer, E.Balaguruswamy Tata McGraw Hill Companies
2. Java Programming John P. Flynt Thomson 2nd
3. Java Programming Language Ken Arnold Pearson
4. The complete reference JAVA2, Herbert schildt. TMH
5. Big Java, Cay Horstmann 2nd edition, Wiley India Edition
6. Core Java, Dietel and Dietel
7. Java - Balaguruswamy
8. Java server programming, Ivan Bayross SPD

Unit-14
AWT

Structure of the Unit

14.0 Objective

This chapter provides a general overview of-

- Abstract Window Toolkit and its classes
- Hierarchy of AWT
- Event handling mechanism
- Various AWT controls

14.1 Introduction

In the last unit we have discussed about applets and their life cycle. We have also discussed various GUI controls that help to make a perfect GUI. In this unit we are focusing on AWT classes, the hierarchy of java.awt package. We are also discussing the Delegation Event Model used by java to handle the events. At last this unit will give a brief overview of Adapter classes and AWT controls.

The classes and interfaces of the Abstract Windowing Toolkit (AWT) are used to develop stand-alone applications and to implement the GUI controls used by applets. These classes support all aspects of GUI development, including event handling.

14.2 Basic Classes in AWT

The Component and Container classes are two of the most important classes in the java.awt package. The Component class is a common super class for all classes that implement GUI controls. The Container class is a subclass of the Component class and can contain other AWT components.

The Window class is a subclass of the Container class that provides a common set of methods for implementing windows. The Window class has two subclasses, Frame and Dialog, that are used to create Window objects. The Frame class is used to create a main application window, and the Dialog class is used to implement dialog boxes.

14.2.1 Graphics Class

The Graphics class provides the framework to perform all graphics related operations within the AWT. In order to draw, a program requires a valid graphics context. Because the Graphics class is an abstract base class, it cannot be instantiated directly. An instance is typically created by a component, and handed to the program as an argument to a component's update() and paint() methods. It plays two different roles-

First, it provides the graphical environment. The graphical environment is information that will affect drawing operations. This includes the background and foreground colors, the font, and the location and dimensions of the component in which graphics can be drawn. It even includes information about the eventual destination of the graphics operations themselves.

Second, this class provides methods for drawing simple geometric shapes (e.g. rectangle, circle, oval, polygon etc), text, and images to the graphics destination. All output to the graphics destination occurs via an invocation of one of these methods.

The following three methods are involved in displaying graphics. Default versions of each are provided by class Component. Methods update() and paint() should be redefined to perform the desired graphics operations.

A) repaint()

```
public void repaint()

public void repaint(long tm)

public void repaint(int x, int y, int w, int h)

public void repaint(long tm, int x, int y, int w, int h)
```

The repaint() method requests that a component be repainted. The caller may request that repainting occur as soon as possible, or may specify a period of time in milliseconds (the second version of method- long tm). If a period of time is specified, the painting operation will occur before the period of time elapses. The caller may also specify that only a portion of a component be repainted (the third version of method). This technique is useful if the paint operation is time consuming, and only a portion of the display needs repainting (the fourth version of method). The following code illustrates how the repaint() method can be used in a program.

```
boolean mouseDown(Event e, int x, int y)

        {

selected_object.move(x, y);

repaint();

 }
```

The code in the mouseDown() event handler recalculates the position of an object and calls the repaint() method to indicate that the display should be repainted as soon as possible.

B) update()

public void update(Graphics g)

The update() method is called in response to a repaint() request, or in response to a portion of the component being uncovered or displayed for the first time. The method's argument is an instance of the Graphics class. The Graphics instance is valid only within the context of the update() method. It is disposed of soon after the update() method returns. The default implementation provided by the Component class erases the background and calls the paint() method.

C) paint()

public void paint(Graphics g)

The paint() method is called from an update() method, and is responsible for actually drawing the graphics. The method's argument is an instance of the Graphics class. The default implementation provided by class Component does nothing.

Repainting of components

To reduce the time required to repaint the display, the AWT takes two shortcuts: First, the AWT repaints only those components that need repainting, either because they have been uncovered, or because they asked to be repainted. Second, if a component was covered and is uncovered, the AWT repaints only the portion of the component that was previously covered.

14.3 Drawing with Graphics Class

14.3.1 The Graphics Coordinate System

Following methods uses the Graphics Coordinate System that take, as parameters, values that specify how a shape is to be drawn. For example, the drawLine() method expects four parameters. The first two parameters specify the location of the beginning of the line, and the last two parameters specify the location of the end of the line. The exact values to be passed to the drawLine() method are determined by the coordinate system in effect.

A coordinate system is a method for unambiguously specifying the location of points in space. In the case of the AWT, this space is a two-dimensional surface called a plane. Each location in a plane can be specified by two integers, called the x and y coordinates. The values of the x and y coordinates are calculated in terms of the point's respective horizontal and vertical displacement from the origin. In the case of the AWT, the origin is always the point in the upper-left corner of the plane. It has the coordinate values 0 (for x) and 0 (for y).

14.3.2 The Graphics Primitives

It includes methods for drawing lines, rectangles, ovals and arcs, and polygons. These methods may be used only within the scope of a component's update() and paint() methods. Method drawX()draws only the outline of the speified shape, and the other method fillX() method draws a filled version of the specified shape.

A) lines

void drawLine(int xBegin, int yBegin, int xEnd, int yEnd)

This is the simplest of all graphics methods. It draws a straight line, a single pixel wide, between the specified beginning and ending points. The resulting line will be clipped to fit within the boundaries of the current clipping region. The line will be drawn in the current foreground color.

B) Rectangles

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

void fillRect(int x, int y, int width, int height)

void drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)

void fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)

void draw3DRect(int x, int y, int width, int height, boolean raised)

void fill3DRect(int x, int y, int width, int height, boolean raised)

Each of these graphics methods require, as parameters, the x and y coordinates at which to begin the rectangle, and the width and the height of the rectangle. Both the width and the height must be positive integers. The resulting rectangle will be clipped to fit within the boundaries of the current clipping region. The rectangle will be drawn in the current foreground color. Rectangles come in three different styles: plain, with rounded corners, and with a slight (but often hard-to-see) three-dimensional effect.

The rounded-rectangle graphics methods require two additional parameters, an arc width and an arc height, both of which control the rounding of the corners. The three-dimensional rectangle methods require an additional parameter that indicates whether or not the rectangle should be sunken or raised.

C) Ovals and Arcs

void drawOval(int x, int y, int width, int height)

void fillOval(int x, int y, int width, int height)

void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)

void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)

These graphics methods require, as parameters, the x and y coordinates of the center of the oval or arc, and the width and height of the oval or arc. Both the width and the height must be positive integers. The resulting shape will be clipped to fit within the boundaries of the current clipping region. The shape will be drawn in the current foreground color.

The arc graphics methods require two additional parameters, a start angle and an arc angle, to specify the beginning of the arc and the size of the arc in degrees (not radians).

D) Polygons

void drawPolygon(int xPoints[], int yPoints[], int nPoints)

void drawPolygon(Polygon p)

void fillPolygon(int xPoints[], int yPoints[], int nPoints)

void fillPolygon(Polygon p)

Polygons are shapes formed from a sequence of line segments. Each of the polygon graphics methods require, as parameters, the coordinates of the endpoints of the line segments that make up the polygon. These endpoints can be specified in either one of two ways: as two parallel arrays of integers, one representing the successive x coordinates and the other representing the successive y coordinates; or with an instance of the Polygon class. The Polygon class provides the method addPoint(), which allows a polygon definition to be assembled point by point. The resulting shape will be clipped to fit within the boundaries of the current clipping region.

14.4 Hierarchy of AWT

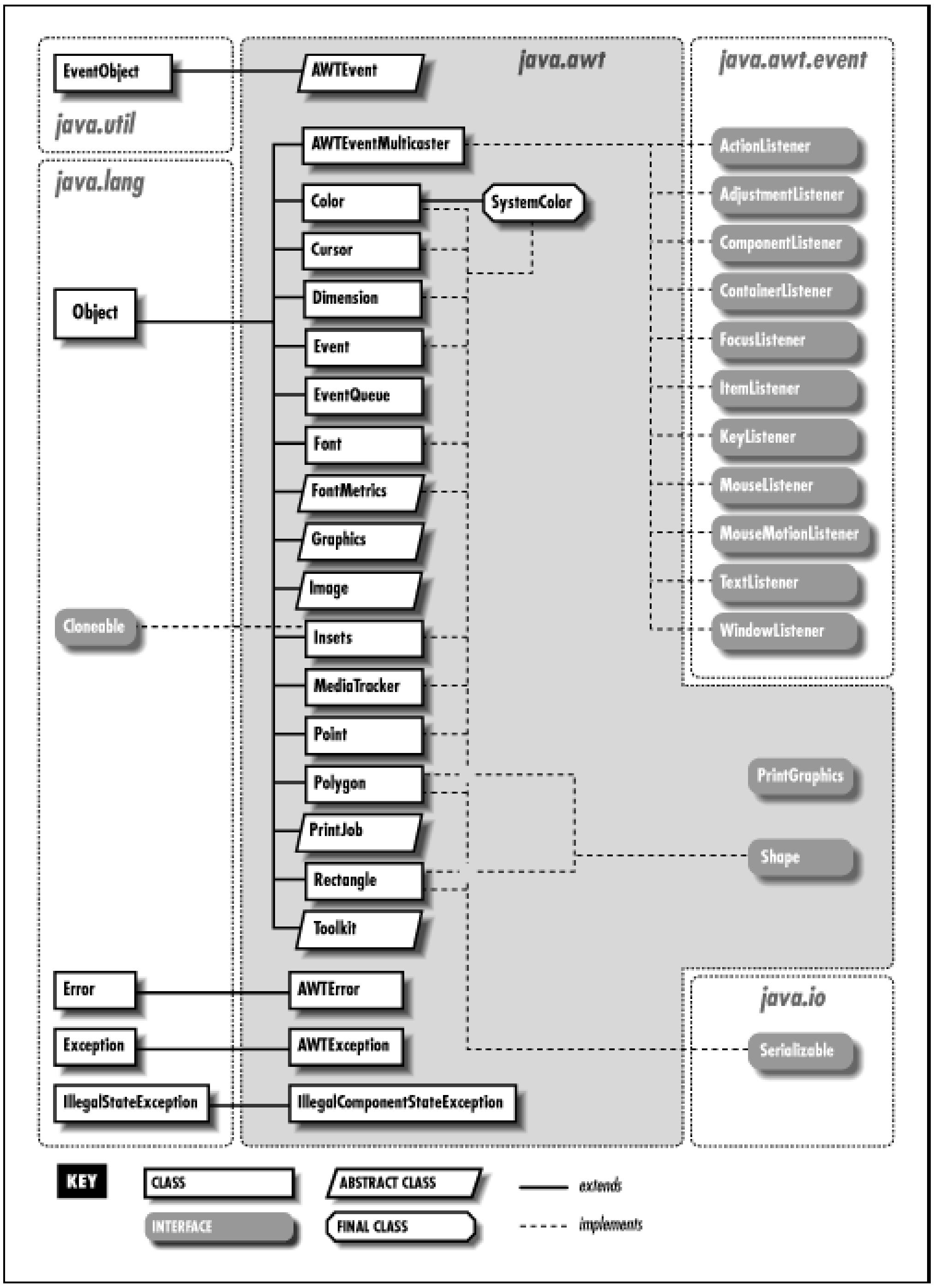


Figure 14.1 Hierarchy of AWT

14.4.1 Types of Containers

The AWT provides four container classes. They are class Window and its two subtypes: class Frame and class Dialog, as well as the Panel class. In addition to the containers provided by the AWT, the Applet class is a container. It is a subtype of the Panel class and can therefore hold components. Brief descriptions of each container class provided by the AWT are provided below.

Window A top-level display surface (a window). An instance of the Window class is not attached to nor embedded within another container. An instance of the Window class has no border and no title.

Frame	A top-level display surface (a window) with a border and title. An instance of the Frame class may have a menu bar. It is otherwise very much like an instance of the Window class.
Dialog	A top-level display surface (a window) with a border and title. An instance of the Dialog class cannot exist without an associated instance of the Frame class.
Panel	A generic container for holding components. An instance of the Panel class provides a container to which to add components.

14.4.2 Creating a Container

Before adding the components that make up a user interface, the programmer must create a container. When building an application, the programmer must first create an instance of class Window or class Frame. When building an applet, a frame (the browser window) already exists. Since the Applet class is a subtype of the Panel class, the programmer can add the components to the instance of the Applet class itself.

The following code creates an empty frame. The title of the frame ("Example 1") is set in the call to the constructor. A frame is initially invisible and must be made visible by invoking its show() method.

```
import java.awt.*;

public class Example1

{

  public static void main(String [] args)

  {

    Frame f = new Frame("Example 1");

    f.setSize(200,300);

    f.show();

  }

}
```

The following code extends the above mentioned code so that the new class inherits from class Panel. In the main() method, an instance of this new class is created and added to the Frame object via a call to the add() method. The result is then displayed. The results of both examples should look identical.

```
import java.awt.*;

public class Example1a extends Panel

{

  public static void main(String [] args)

  {

    Frame f = new Frame("Example 1a");

    Example1a ex = new Example1a();

    f.add("Center", ex);
```

```
      f.pack();
     f.show();
   }
}
```

By deriving the new class from class Applet instead of class Panel, this example can now run as either a standalone application or as an applet embedded in a Web page.

```
import java.awt.*;
import java.applet.*;
/*<applet code="Exampleb" height=400 width=200>
</applet>
public class Example1b extends Applet
{
  public static void main(String [] args)
   {
     Frame f = new Frame("Example 1b");
     Example1b ex = new Example1b();
     f.add("Center", ex);
      f.pack();
     f.show();
   }
}
```

Note: a Window object, and in certain cases even a Dialog object, could replace the Frame object. They are all valid containers, and components are added to each in the same fashion.

14.4.3 Adding Components to a Container

To be useful, a user interface must consist of more than just a container. It must contain components. Components are added to containers via a container's add() method. Following code creates and adds two buttons on to container. The creation is performed in the init() method because it is automatically called during applet initialization. Therefore, no matter how the program is started, the buttons are created, because init() is called by either the browser or by the main() method.

```
import java.awt.*;
import java.applet.*;
public class Example3 extends Applet
{
   public void init()
   {
      add(new Button("One"));
```

```
    add(new Button("Two"));
   }
  public Dimension preferredSize()
   {
    return new Dimension(200, 100);
   }
  public static void main(String [] args)
   {
    Frame f = new Frame("Example 3");
    Example3 ex = new Example3();
    ex.init();
    f.add("Center", ex);
     f.pack();
     f.show();
   }
}
```

14.5 Event Handling

Event Handling can be done through the java.awt package of java. Events are the integral part of the java platform. AwtEvent is the main class of the program which extends from the Frame class implements the ActionListener interface.

14.5.1 Delegation Event Model

This approach defines standard and consistent mechanism to generate and process the events. The concept is as follows-

A source generates an event and sends it to one or more listeners that are registered. Once received the listener processes the event and then returns. The advantage of this design is that application logic is separated from user interface logic that generates the events. The user interface element delegates the processing of an event to a separate piece of code. An event is an object that describes change in the state of an item (source). Event Source is an object that generates the events. It occurs when internal state of an object changes in some way. A source may be registered by following code-

```
public void addTypeListener(TypeListener el)
```

Type is the name of the event and el is the reference of event listener e.g. addKeyListener() will register keyboard events, addMouseMotionListener() will register mouse motion events.

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 notifications. Second it must implement methods to receive and process these notifications. Table 14.1 describes event classes.

Table 14.1: Main Event classes in java.awt.event.

Event Class	**Description**
ActionEvent	Generated when a button is presses, a list item is double-clicked, or a menu item is selected
AdjustmentEvent	Generated when scroll bar is manipulated
ComponentEvent	Generated when a component is hidden, moved, resized or becomes visible
ContainerEvent	Generated when a component is added to removed from a container
FocusEvent	Generated when a component gains or loses keyboard focus
InputEvent	Abstract super class for all component input event class
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or checkable menu item is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed or released; also generated when the mouse enters or exits a component
MouseWheelEvent	Generated when the mouse wheel is moved
TextEvent	Generated when the value of a text area or text field is changed
WindowEvent	Generated when a window is activated, deactivated, closed, deiconified, iconified, opened or quit.

Table 14.2 gives a brief overview of sources of events. It may be a clicking a button, selecting a checkbox, selecting an item from the list etc.

Table 14.2: Sources of Events

Event Source	**Description**
Button	Generates action events when the button is pressed
Checkbox	Generates item events when the checkbox is selected or deselected
Choice	Generates item events when the choice is changed
List	Generates action events when item is double-clicked; generates item events when 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, opened, deiconified, iconified, deactivated or quit.

Table 14.2 gives a brief overview of various event listener interfaces. The methods of these interfaces are used when appropriate event is occurred.

Table 14.3: Event Listener Interfaces

Interface	Description
ActionListener	Defines one method to receive action events
AdjustmentListener	Defines one method to receive adjustment events
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized or shown
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus
ItemListener	Defines one method to recognize when the state of an item is changed
KeyListener	Defines three methods to recognize when a key is pressed, released or typed
MouseListener	Defines five methods to recognize when the mouse is clicked, pressed, released, enters a component, exits a component
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved
TextListener	Defines one method to recognize when a text value changes
WindowFocusListener	Defines two methods to recognize when window gains or loses input focus
WindowListener	Defines seven methods to recognize when window is activated, deactivated, iconified, deiconifoed, opened, closed, or quit

Following program demonstrates what happens when an event is occurred. This program starts to run from the main method in which the object for the AwtEvent class has been created. The constructor of the AwtEvent class creates two buttons with adding the addActionListener() method to it. The constructor also initializes a label with text "VMOU Kota".

When you click on the button then the actionPerformed() method is called which receives the generated event. This method shows the text of the source of the event on the label.

Here is the code of the program :

```
import java.awt.*;

import java.awt.event.*;

public class AwtEvent extends Frame implements ActionListener{

  Label lbl;

  public static void main(String argv[]){

  AwtEvent t = new AwtEvent();

  }
```

```
 public AwtEvent(){
 super("Event in Java awt");
 setLayout(new BorderLayout());
 try{
 Button button = new Button("Button1");
 button.addActionListener(this);
 add(button, BorderLayout.NORTH);
 Button button1 = new Button("Button2");
 button1.addActionListener(this);
 add(button1, BorderLayout.SOUTH);
 lbl = new Label("VMOU Kota");
 add(lbl, BorderLayout.CENTER);
 addWindowListener(new WindowAdapter(){
 public void windowClosing(WindowEvent we){
 System.exit(0);
  }
  });
  }
 catch (Exception e){}
 setSize(400,400);
 setVisible(true);
  }
 public void actionPerformed(ActionEvent e){
 Button bt = (Button)e.getSource();
 String str = bt.getLabel();
 lbl.setText(str);
  }
}
```

14.5.2 Adapter Classes

In java programming language, adapter class is used to implement an interface having a set of dummy methods. It provides an empty implementation of all methods in an event listener interface. These classes are useful when we want to receive and process only some of the events that are handled by a particular event listener interface. Examples are

MouseAdapter

MouseMotionAdapter

The developer can then further subclass the adapter class so that he can override to the methods he requires. Implementing an interface directly requires writing all the dummy methods. In general an adapter class is used to rapidly construct your own Listener class to field events.

Table 14.4 shows the list of adapter classes corresponding to listener interfaces

Table 14.4 Listener Interfaces implemented by Adapter Classes

Adapter Class	**Listener Interface**
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

Following code demonstrates an example

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/* <applet code="AdapterDemo" height=400 width=400>
</applet> */
public class AdapterDemo extends Applet{
 public void init(){
 addMouseListener(
 new MouseAdapter(){
 int topX, bottomY;
 public void mousePressed(MouseEvent me){
 topX = me.getX();
 bottomY = me.getY();
 }
 public void mouseReleased(MouseEvent me){
 Graphics g = AdapterDemo.this.getGraphics();
 g.drawRect(topX, bottomY, me.getX()-topX, me.getY()-bottomY);
 }
 });
 }
}
```

When you compile and run this program, you will have the following results

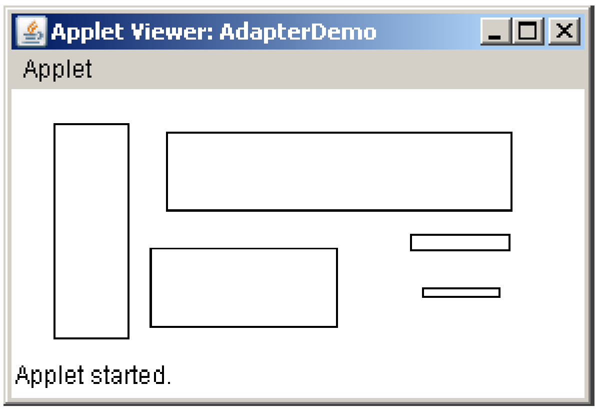


Figure 14.2: Applet using Adapter Class

14.6 AWT Controls

The AWT provides nine basic non-container component classes from which a user interface may be constructed. These nine classes are class Button, Canvas, Checkbox, Choice, Label, List, Scrollbar, TextArea, and TextField.

A) Labels

Constructors

Label()

Label(String str)

Label(String str, int how)

Methods

void setText(String str)

String getText()

void setAlignment(int how)

int getAlignment()

B) Buttons

Constructors

Button()

Button(String str)

Methods

void setLabel(String str)

String getLabel()

C) Check boxes

Constructors

Checkbox()

Checkbox(String str)

Checkbox(String str, boolean on)

Checkbox(String str, boolean on, CheckboxGroup cbGroup)

Checkbox(String str, CheckboxGroup cbGroup, boolean on)

Methods

boolean getState()

void setState(boolean on)

String getLabel()

void setLabel(String str)

Checkbox getSelectedCheckbox()

void setSelectedCheckbox(Checkbox which)

D) Choice

Constructors

List()

List(int numRows)

List(int numRows, boolean multipleSelect)

Methods

void add(String str)

void add(String str, int index)

String getSelectedItem()

int getSelectedIndex()

String[] getSelectedItems()

Int[] getSelectedIndexes()

int getItemCount()

void select(int index)

String getItem(int index)

E) Scroll bars

Constructors

Scrollbar()

Scrollbar(int style)

Scrollbar(int style, int initialValue, int thumbSize, int min, int max)

Methods

void setValues(int initialValue, int thumbSize, int min, int max)

int getValue()

void setValue(int newValue)

int getMinimum()

int getMaximum()

void setUnitIncrement(int newIncr)

void setBlockIncrement(int newIncr)

F) TextField

Constructors

TextField()

TextField(int numChars)

TextField(String str)

TextField(String str, int numChars)

Methods

String getText()

void setText(String str)

String getSelectedText()

void select(int startIndex, int endIndex)

boolean isEditable()

void setEditable(boolean canEdit)

void setEchoChar(char ch)

boolean echoCharIsSet()

char getEchoChar()

G) TextArea

Constructors

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)

Methods

String getText()

void setText(String str)

String getSelectedText()

void select(int startIndex, int endIndex)

boolean isEditable()

void setEditable(boolean canEdit)

void append(String str)

void insert(String str, int index)

void replaceRange(String str, int index, int endIndex)

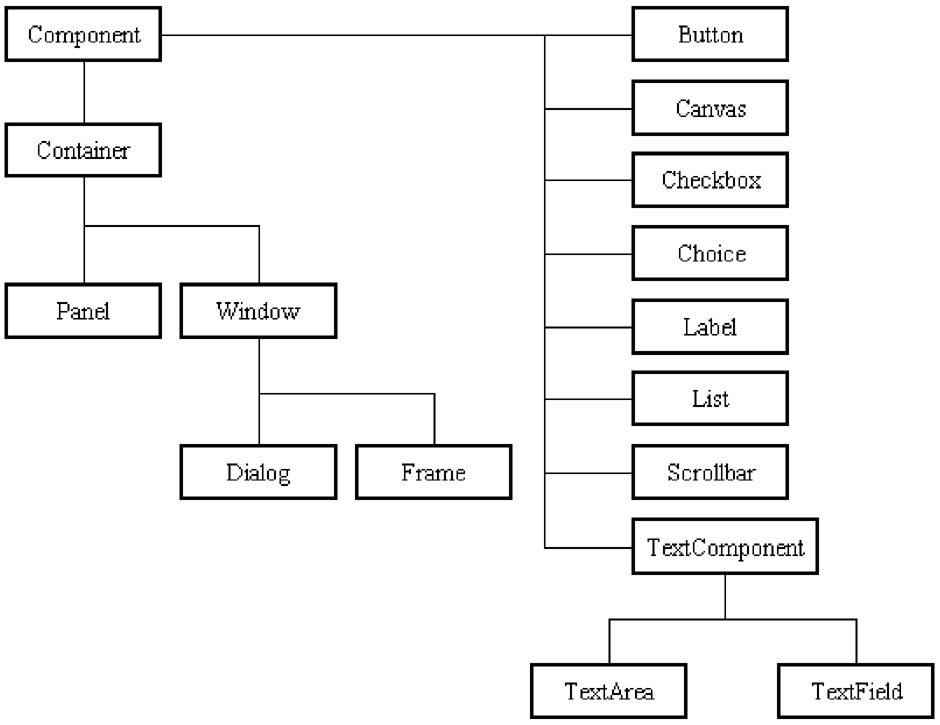


Figure 14.3. The inheritance relationship

14.7 Summary

In this unit we have discussed the package awt, its classes and hierarchy. We have also discussed two event handling models i.e. Delegation Event Model and handling events using Adapter classes. Finally we have discussed nine AWT controls, their constructor definitions and their methods.

14.8 Self-Assessment Questions

1. Write an applet that will accept a text from a keyboard & display it on the screen as a moving text from left to right. Use key event to input the text.
2. Write a program to implement mouseClicked method using MouseAdapter Class.
3. Write an applet that will draw a line, rectangle, oval arc, round rectangle, and filled rectangle in different colors.

14.9 References

1. Programming with Java A Primer, E.Balaguruswamy Tata McGraw Hill Companies
2. Java Programming John P. Flynt Thomson 2nd
3. Java Programming Language Ken Arnold Pearson
4. The complete reference JAVA2, Herbert schildt. TMH
5. Big Java, Cay Horstmann 2nd edition, Wiley India Edition
6. Core Java, Dietel and Dietel
7. Java - Balaguruswamy
8. Java server programming, Ivan Bayross SPD

Unit-15
Swing in Java

Structure of the Unit

15.0 Objective

This chapter provides a general overview of-

- Layout Managers
- Swings
- Advanced Layout Managers
- Additional Swing Components

15.1 Introduction

In the previous chapter we discussed about the various Controls in Java like Button, Checkbox, Lists, Scrollbars, Text Fields, and Text Area etc. All of these components have been positioned by the default layout manager. A layout manager automatically arranges the controls within a window by using some type of algorithm.

15.2 Layout Managers

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 each of the components within it.

The setLayout( ) method has the following general form:

void setLayout(LayoutManager layoutObj)

Here, layoutObj is a reference to the desired layout manager. If you wish to disable the layout manager and position components manually, pass null for layoutObj. If you do this, you will need to determine the shape and position of each component manually, using the setBounds( ) method defined by Component.

Each layout manager keeps track of a list of components that are stored by their names. The layout manager is notified each time you add a component to a container. Normally, you will want to use a layout manager. Whenever the container needs to be resized, the layout manager is consulted via its minimumLayoutSize( ) and preferredLayoutSize( ) methods. Each component that is being managed by a layout manager contains the getPreferredSize( ) and getMinimumSize( ) methods. These return the preferred and minimum size required to display each component. You may override these methods for controls that you subclass. Default values are provided otherwise.

Java has several predefined LayoutManager classes.

- FlowLayout
- BorderLayout
- GridLayout
- CardLayout

15.2.1 FlowLayout Manager

FlowLayout is the default layout manager. FlowLayout implements a simple layout style, which is similar to how words flow in a text editor. Components are laid out from the upper-left corner, left to right and top to bottom. When no more components fit on a line, the next one appears on the next line. A small space is left between each component, above and below, as well as left and right.

The constructors for FlowLayout are shown below:

1. FlowLayout( )
2. FlowLayout(int how)
3. FlowLayout(int how, int horz, int vert)

The first form creates the default layout, which centers components and leaves five pixels of space between each component.

The second form lets you specify how each line is aligned. Valid values for how are as follows:

- FlowLayout.LEFT
- FlowLayout.CENTER
- FlowLayout.RIGHT

These values specify left, center, and right alignment, respectively. The third form allows specifying the horizontal and vertical space left between components in horz and vert, respectively.

Example

```
import java.awt.*;

class flow extends Frame

{
        flow(String title, int count)
        {
                super(title);
                setLayout(new FlowLayout());
                for(int i = 0 ;i<count;i++)
                add( new Button(Integer.toString(i)));
        }
        public static void main(String args[])
        {
        int count=(args.length==0)?5:Integer.parseInt(args[0]);
        flow flow1= new flow("first frame",count);
        flow1.setSize(200,300);
        flow1.show();
        }
}
```

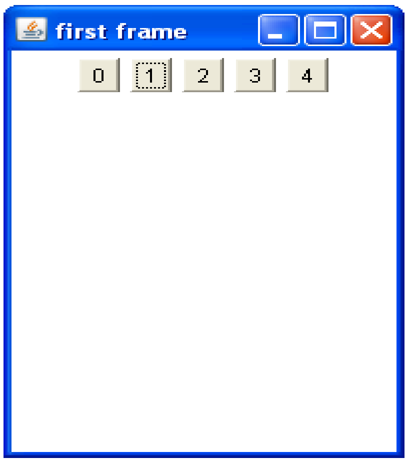


Figure 15.1: Demo of Flow Layout Manager

15.2.2 BorderLayout Manager

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 defined by BorderLayout are shown below:

1. BorderLayout( )
2. BorderLayout(int horz, int vert)

The first form creates a default border layout.

The second allows you to specify the horizontal and vertical space left between components in horz and vert, respectively.

BorderLayout defines the following constants that specify the regions:

- BorderLayout.CENTER
- BorderLayout.SOUTH
- BorderLayout.EAST
- BorderLayout.WEST
- BorderLayout.NORTH

When adding components, you will use these constants with the following form of add( ), which is defined by Container:

void add(Component compObj, Object region)

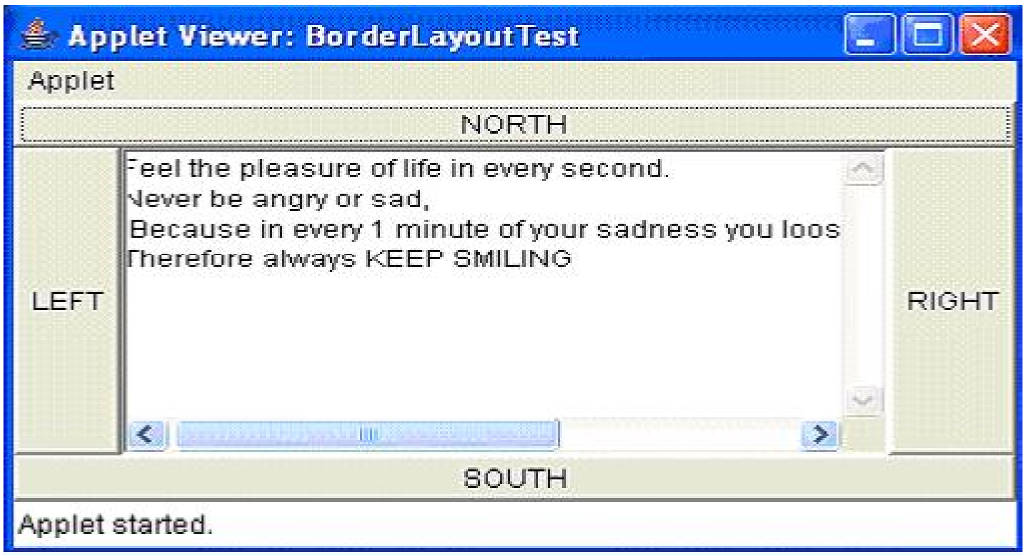
Here, compObj is the component to be added, and region specifies where the component will be added.

Example

Here is an example of a BorderLayout with a component in each layout area:

```
import java.awt.*;
import java.applet.*;
import java.util.*;
/*
<applet code="BorderLayoutTest" width=400 height=200>
</applet>
*/
public class BorderLayoutTest extends Applet
{
   public void init()
    {
      setLayout(new BorderLayout());
      add(new Button("NORTH"), BorderLayout.NORTH);
      add(new Button("SOUTH"), BorderLayout.SOUTH);
      add(new Button("RIGHT"), BorderLayout.EAST);
      add(new Button("LEFT"), BorderLayout.WEST);
      String str = "Feel the pleasure of life in every second.\n" +
       "Never be angry or sad, \n " +
      "Because in every 1 minute of your sadness " +
      "you loose 60 seconds of your hapiness.\n" +
      "Therefore always KEEP SMILING \n\n";
      add(new TextArea(str), BorderLayout.CENTER);
    }
}
```

Output would be as shown below:



```
Feel the pleasure of life in every second.
Never be angry or sad,
Because in every 1 minute of your sadness you loos
Therefore always KEEP SMILING
```

Applet started.

Figure 15.2: Demo of Border Layout Manager

Insets

Sometimes you will want to leave a small amount of space between the container that holds your components and the window that contains it. To do this, override the getInsets( ) method that is defined by Container. This function returns an Insets object that contains the top, bottom, left, and right inset to be used when the container is displayed. These values are used by the layout manager to inset the components when it lays out the window.

The constructor for Insets is Insets(int top, int left, int bottom, int right)

The values passed in top, left, bottom, and right specify the amount of space between the container and its enclosing window.

The getInsets( ) method has this general form:

Insets getInsets( )

When overriding one of these methods, you must return a new Insets object that contains the inset spacing you desire.

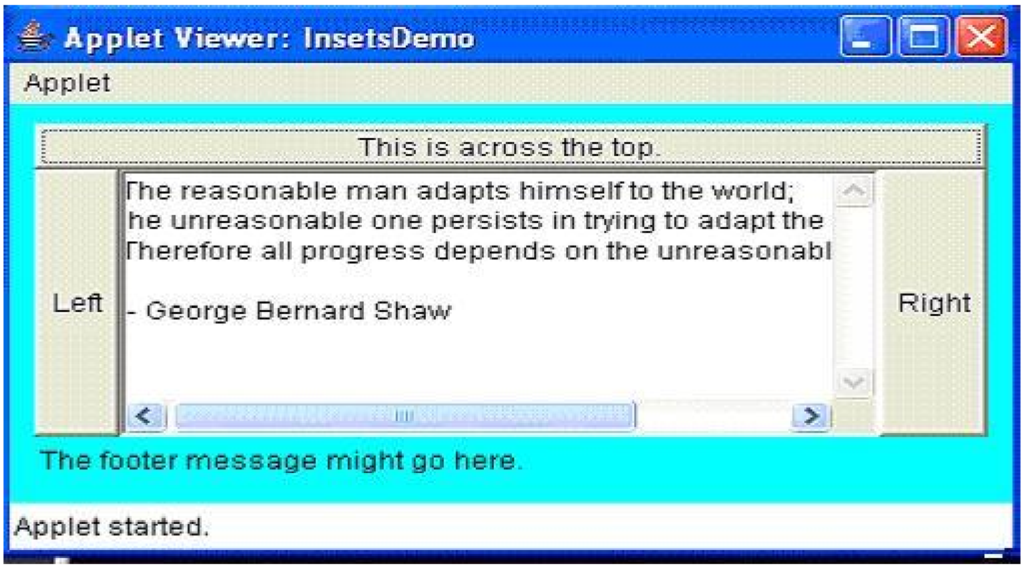
Example:

Here is the preceding BorderLayout example modified so that it insets its components ten pixels from each border. The background color has been set to cyan to help make the insets more visible.

```
// Demonstrate BorderLayout with insets.

import java.awt.*;

import java.applet.*;

import java.util.*;
```

```
/*
<applet code="InsetsDemo" width=400 height=200>
</applet>
*/
public class InsetsDemo extends Applet
{
  public void init()
   {
     // set background color so insets can be easily seen
     setBackground(Color.cyan);
     setLayout(new BorderLayout());
     add(new Button("This is across the top."),
     BorderLayout.NORTH);
     add(new Label("The footer message might go here."),
     BorderLayout.SOUTH);
     add(new Button("Right"), BorderLayout.EAST);
     add(new Button("Left"), BorderLayout.WEST);
     String msg = "The reasonable man adapts " +
     "himself to the world;\n" +
     "the unreasonable one persists in " +
     "trying to adapt the world to himself.\n" +
     "Therefore all progress depends " +
     "on the unreasonable man.\n\n" +
     " - George Bernard Shaw\n\n";
     add(new TextArea(msg), BorderLayout.CENTER);
   }
     // add insets
  public Insets getInsets()
   {
      return new Insets(10, 10, 10, 10);
   }
}
```

Output would be as shown below:



The reasonable man adapts himself to the world;
he unreasonable one persists in trying to adapt the
Therefore all progress depends on the unreasonabl

- George Bernard Shaw

Figure 15.3: Demo of Border Layout Manager with insets

15.2.3 GridLayout Manager

GridLayout lays out components in a two-dimensional grid. When you instantiate a GridLayout, you define the number of rows and columns.

The constructors supported by GridLayout are shown below:

1. GridLayout( )
2. GridLayout(int numRows, int numColumns)
3. GridLayout(int numRows, int numColumns, int horz, int vert)

The first form creates a single-column grid layout.

The second form creates a grid layout with the specified number of rows and columns. The third form allows you to specify the horizontal and vertical space left between components in horz and vert, respectively. Either numRows or numColumns can be zero. Specifying numRows as zero allows for unlimited-length columns. Specifying numColumns as zero allows for unlimited-length rows.

Example

```
import java.awt.*;

class GridLay extends Frame

{

        GridLay(String title, int rows, int cols)

        {

                super(title);

                setLayout(new GridLayout(rows, cols));

                for(int i=0;i<rows;i++)

                        for(int j=0;j<cols;j++)
```

```
                    {
                    if(i==j)
                    continue;
                    add(new Button(i+","+j));
                    }
        }
        public static void main(String args[])
        {
        GridLay gl1= new GridLay("first frame",4,4);
        gl1.setSize(200,300);
        gl1.show();
        }
}
```

The output would be as shown below-

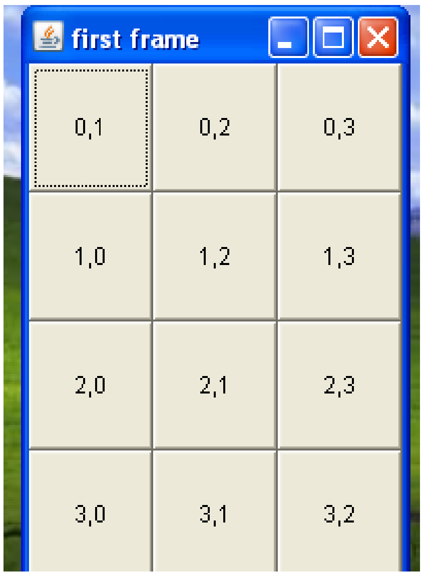


Figure 15.4: Demo of Grid Layout Manager

15.2.4 CardLayout Manager

The CardLayout class is unique among the other layout managers in that it 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. This can be useful for user interfaces with optional components that can be dynamically enabled and disabled upon user input. We can prepare the other layouts and have them hidden, ready to be activated when needed.

CardLayout provides the following two constructors:

1. CardLayout( )
2. CardLayout(int horz, int vert)

The first form creates a default card layout.

The second form allows you to specify the horizontal and vertical space left between components in horz and vert, respectively.

Use of a card layout requires a bit more work than the other layouts. The cards are typically held in an object of type Panel. This panel must have CardLayout selected as its layout manager. The cards that form the deck are also typically objects of type Panel.

Thus, you must create a panel that contains the deck and a panel for each card in the deck. Next, you add to the appropriate panel the components that form each card. You then add these panels to the panel for which CardLayout is the layout manager. Finally, you add this panel to the main applet panel. Once these steps are complete, you must provide some way for the user to select between cards. One common approach is to include one push button for each card in the deck.

When card panels are added to a panel, they are usually given a name. Most of the time, you will use this form of add( ) when adding cards to a panel:

void add(Component panelObj, Object name)

Here, name is a string that specifies the name of the card whose panel is specified by panelObj.

After you have created a deck, your program activates a card by calling one of the following methods defined by CardLayout:

1. void first(Container deck)
2. void last(Container deck)
3. void next(Container deck)
4. void previous(Container deck)
5. void show(Container deck, String cardName)

Here, deck is a reference to the container (usually a panel) that holds the cards, and cardName is the name of a card.

Calling first( ) causes the first card in the deck to be shown.

To show the last card, call last( ).

To show the next card, call next( ).

To show the previous card, call previous( ).

Both next( ) and previous( ) automatically cycle back to the top or bottom of the deck, respectively.

The show( ) method displays the card whose name is passed in cardName.

Example:

The following example creates a two-level card deck that allows the user to select a language. Procedural languages are displayed in one card. Object Oriented languages are displayed in the other card.

```
// Demonstrate CardLayout.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CardLayoutTest" width=400 height=100>
</applet>
```

```
*/
public class CardLayoutTest extends Applet implements ActionListener, MouseListener
{
  Checkbox Java, C, VB ;
  Panel langCards;
  CardLayout cardLO;
  Button OO, Other;
  public void init()
   {
    OO = new Button("ObjectOriented Languages");
    Other = new Button("Procedural Languages");
     add(OO);
     add(Other);
     cardLO = new CardLayout();
     langCards = new Panel();
    langCards.setLayout(cardLO); // set panel layout to card layout
    Java = new Checkbox("Java", null, true);
     C = new Checkbox("C");
     VB = new Checkbox("VB");
     // add OO languages check boxes to a panel
     Panel OOPan = new Panel();
     OOPan.add(Java);
     OOPan.add(VB);
    // Add other languages check boxes to a panel
     Panel otherPan = new Panel();
     otherPan.add(C);
     // add panels to card deck panel
    langCards.add(OOPan, "Object Oriented Languages");
    langCards.add(otherPan, "Procedural Languages");
     // add cards to main applet panel
     add(langCards);
    // register to receive action events
    OO.addActionListener(this);
    Other.addActionListener(this);
    // register mouse events
    addMouseListener(this);
   }
  // Cycle through panels.
  public void mousePressed(MouseEvent me)
   {
    cardLO.next(langCards);
   }
  // Provide empty implementations for the 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() == OO)
       {
        cardLO.show(langCards, "Object Oriented Languages");
       }
      else
       {
        cardLO.show(langCards, "Procedural Languages");
       }
   }
}
```

Output would be as shown below:

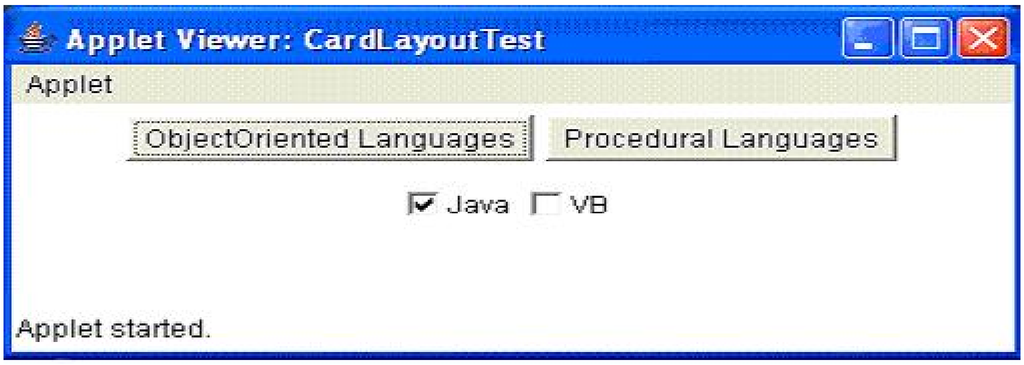


Figure 15.5: Demo of Card Layout Manager-1

On clicking the Procedural languages Button the following output would be displayed:

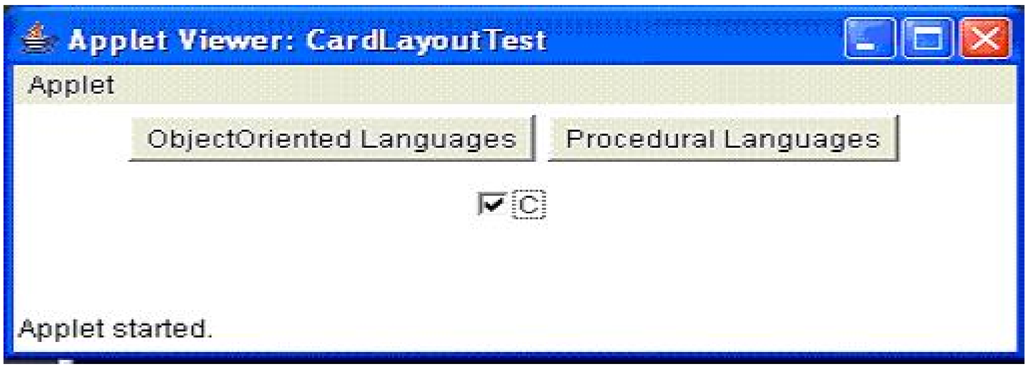


Figure 15.6: Demo of Card Layout Manager-2

15.3 Java Swings

Swing library is an official Java GUI toolkit released by Sun Microsystems. It is used to create Graphical user interfaces with Java.

The main characteristics of the Swing toolkit

- platform independent
- customizable
- extensible
- configurable
- lightweight

Swing is an advanced GUI toolkit. It has a rich set of widgets. From basic widgets like buttons, labels, scrollbars to advanced widgets like trees and tables. Swing itself is written in Java.

Swing is a part of JFC, Java Foundation Classes. It is a collection of packages for creating full featured desktop applications. JFC consists of AWT, Swing, Java 2D, and Drag and Drop. Swing was released in 1997 with JDK 1.2. It is a mature toolkit.

15.3.1 Swings Packages

The Swing API has 18 public packages:

- javax.accessibility
- javax.swing
- javax.swing.border
- javax.swing.colorchooser
- javax.swing.event
- javax.swing.filechooser
- javax.swing.plaf
- javax.swing.plaf.basic
- javax.swing.plaf.metal
- javax.swing.plaf.multi
- javax.swing.plaf.synth
- javax.swing.table
- javax.swing.text

- javax.swing.text.html
- javax.swing.text.html.parser
- javax.swing.text.rtf
- javax.swing.tree
- javax.swing.undo

15.3.2 Hierarchy of Swing

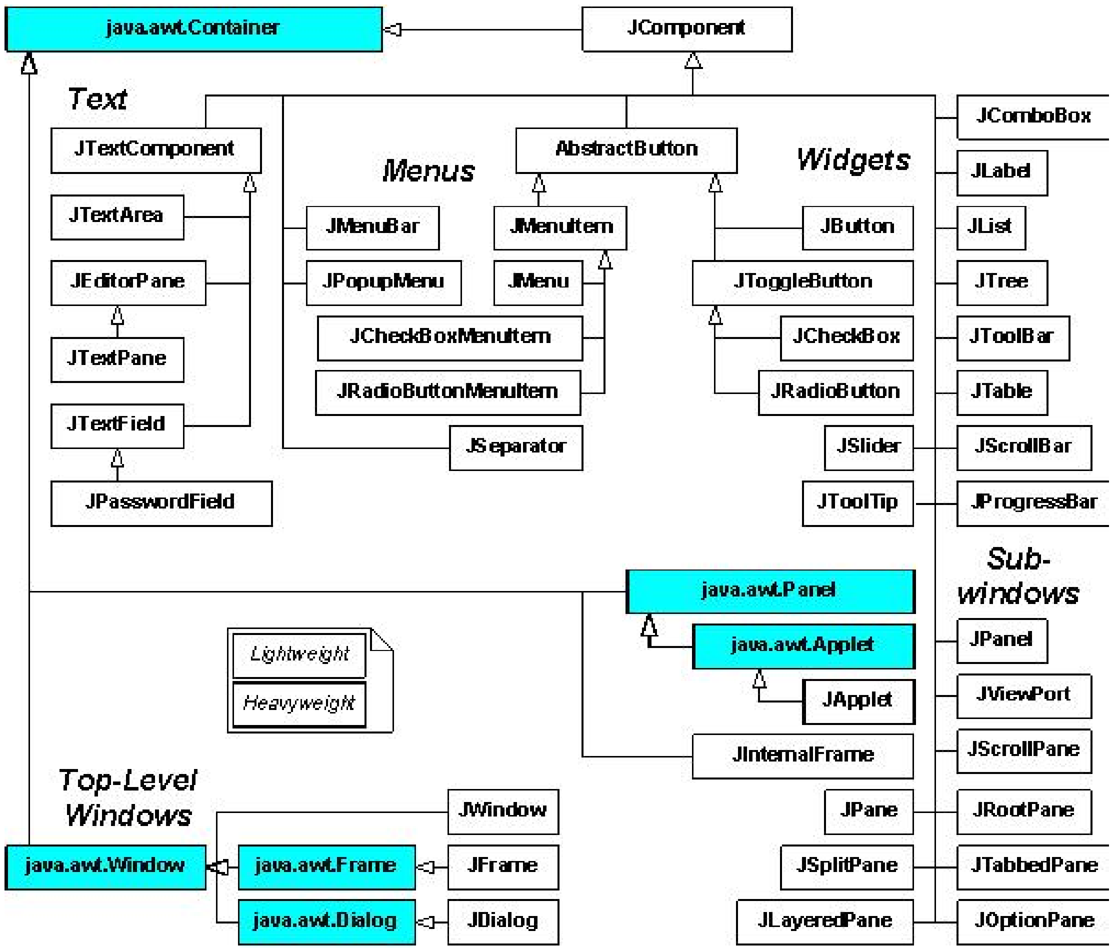


Figure 15.7 Hierarchy of Swing package

15.4 Advanced Layout Managers

The Swing packages include a general purpose layout manager named BoxLayout. BoxLayout either stacks its components on top of each other or places them in a row. You might think of it as a version of FlowLayout, but with greater functionality. Here is a picture of an application that demonstrates using BoxLayout to display a centered column of components:

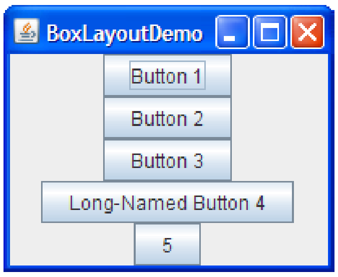


Figure 15.8: Box Layout Manager

15.4.1 Box Layout Manager

BoxLayout arranges components either on top of each other or in a row. As the box layout arranges components, it takes the components' alignments and minimum, preferred, and maximum sizes into account. In this section, we will talk about top-to-bottom layout. The same concepts apply to left-to-right or right-to-left layout. You simply substitute X for Y, height for width, and so on.

When a BoxLayout lays out components from top to bottom, it tries to size each component at the component's preferred height. If the vertical space of the layout does not match the sum of the preferred heights, then BoxLayout tries to resize the components to fill the space. The components either grow or shrink to fill the space, with BoxLayout honoring the minimum and maximum sizes of each of the components. Any extra space appears at the bottom of the container.

For a top-to-bottom box layout, the preferred width of the container is that of the maximum preferred width of the children. If the container is forced to be wider than that, BoxLayout attempts to size the width of each component to that of the container's width (minus insets). If the maximum size of a component is smaller than the width of the container, then X alignment comes into play.

The X alignments affect not only the components' positions relative to each other, but also the location of the components (as a group) within their container. The following figures illustrate alignment of components that have restricted maximum widths.

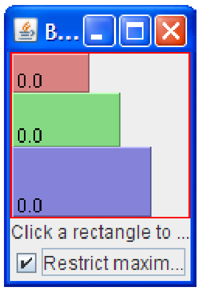


Figure 15.9: Box Layout Manager-1

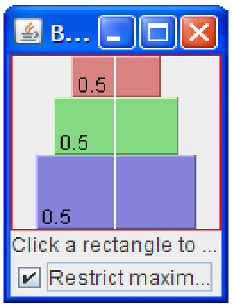


Figure 15.10: Box Layout Manager-2

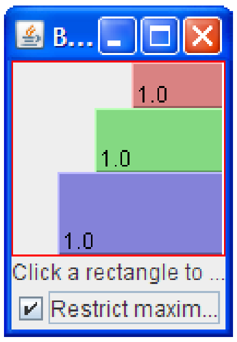


Figure 15.11: Box Layout Manager-3

In the first figure, all three components have an X alignment of 0.0 (Component.LEFT_ALIGNMENT). This means that the components' left sides should be aligned. Furthermore, it means that all three components are positioned as far left in their container as possible.

In the second figure, all three components have an X alignment of 0.5 (Component.CENTER_ALIGNMENT). This means that the components' centers should be aligned, and that the components should be positioned in the horizontal center of their container.

In the third figure, the components have an X alignment of 1.0 (Component.RIGHT_ALIGNMENT). You can guess what that means for the components' alignment and position relative to their container.

15.4.2 Grid BagLayout Manager

GridBagLayout is one of the most flexible and complex layout managers the Java. A GridBagLayout places components in a grid of rows and columns, allowing specified components to span multiple rows or columns. Not all rows necessarily have the same height. Similarly, not all columns necessarily have the same width. Essentially, GridBagLayout places components in rectangles (cells) in a grid, and then uses the components' preferred sizes to determine how big the cells should be.

The following figure shows the grid for the preceding applet. As you can see, the grid has three rows and three columns. The button in the second row spans all the columns; the button in the third row spans the two right columns.

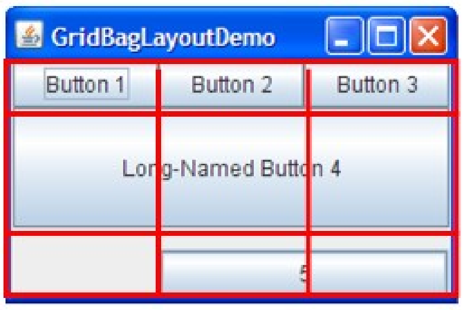


Figure 15.12: Grid Bag Layout Manager-1

If you enlarge the window as shown in the following figure, you will notice that the bottom row, which contains Button 5, gets all the new vertical space. The new horizontal space is split evenly among all the columns. This resizing behavior is based on weights the program assigns to individual components in the GridBagLayout. You will also notice that each component takes up all the available horizontal space but not (as you can see with button 5) all the available vertical space. This behavior is also specified by the program.

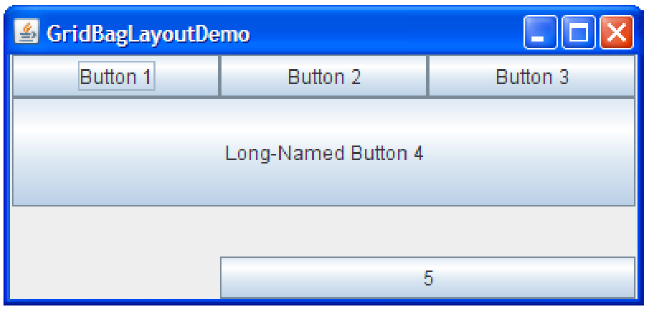


Figure 15.13: Grid Bag Layout Manager-2

The way the program specifies the size and position characteristics of its components is by specifying constraints for each component. The preferred approach to set constraints on a component is to use the Container.add variant, passing it a GridBagConstraints object, as demonstrated in the next sections.

The following sections explain the constraints you can set and provide examples.

Specifying Constraints

The following code is typical of what goes in a container that uses a GridBagLayout. You will see a more detailed example in the next section.

```
JPanel pane = new JPanel(new GridBagLayout());

GridBagConstraints c = new GridBagConstraints();

//For each component to be added to this container:

//...Create the component...

//...Set instance variables in the GridBagConstraints instance...

pane.add(theComponent, c);
```

It is possible to reuse the same GridBagConstraints instance for multiple components, even if the components have different constraints. However, it is recommended that you do not reuse GridBagConstraints, as this can very easily lead to you introducing subtle bugs if you forget to reset the fields for each new instance.

Note: The following discussion assumes that the GridBagLayout controls a container that has a left-to-right component orientation.

You can set the following GridBagConstraints instance variables:

gridx, gridy

Specify the row and column at the upper left of the component. The leftmost column has address gridx=0 and the top row has address gridy=0. Use GridBagConstraints.RELATIVE (the default value) to specify that the component be placed just to the right of (for gridx) or just below (for gridy) the component that was added to the container just before this component was added. We recommend specifying the gridx and gridy values for each component rather than just using GridBagConstraints.RELATIVE; this tends to result in more predictable layouts.

gridwidth, gridheight

Specify the number of columns (for gridwidth) or rows (for gridheight) in the component's display area. These constraints specify the number of cells the component uses, not the number of pixels it uses. The default value is 1. Use GridBagConstraints.REMAINDER to specify that the component be the last one in its row (for gridwidth) or column (for gridheight). Use GridBagConstraints.RELATIVE to specify that the component be the next to last one in its row (for gridwidth) or column (for gridheight). We recommend specifying the gridwidth and gridheight values for each component rather than just using GridBagConstraints.RELATIVE and GridBagConstraints.REMAINDER; this tends to result in more predictable layouts.

Note: GridBagLayout does not allow components to span multiple rows unless the component is in the leftmost column or you have specified positive gridx and gridy values for the component.

fill

Used when the component's display area is larger than the component's requested size to determine whether and how to resize the component. Valid values (defined as GridBagConstraints constants) include NONE (the default), HORIZONTAL (make the component wide enough to fill its display area horizontally, but do not change its height), VERTICAL (make the component tall enough to fill its display area vertically, but do not change its width), and BOTH (make the component fill its display area entirely).

ipadx, ipady

Specifies the internal padding: how much to add to the size of the component. The default value is zero. The width of the component will be at least its minimum width plus ipadx*2 pixels, since the padding applies to both sides of the component. Similarly, the height of the component will be at least its minimum height plus ipady*2 pixels.

insets

Specifies the external padding of the component -- the minimum amount of space between the component and the edges of its display area. The value is specified as an Insets object. By default, each component has no external padding.

anchor

Used when the component is smaller than its display area to determine where (within the area) to place the component. Valid values (defined as GridBagConstraints constants) are CENTER (the default), PAGE_START, PAGE_END, LINE_START, LINE_END, FIRST_LINE_START, FIRST_LINE_END, LAST_LINE_END, and LAST_LINE_START.

weightx, weighty

Specifying weights is an art that can have a significant impact on the appearance of the components a GridBagLayout controls. Weights are used to determine how to distribute space among columns (weightx) and among rows (weighty); this is important for specifying resizing behavior.

Unless you specify at least one non-zero value for weightx or weighty, all the components clump together in the center of their container. This is because when the weight is 0.0 (the default), the GridBagLayout puts any extra space between its grid of cells and the edges of the container.

Generally weights are specified with 0.0 and 1.0 as the extremes: the numbers in between are used as necessary. Larger numbers indicate that the component's row or column should get more space. For each column, the weight is related to the highest weightx specified for a component within that column, with each multicolumn component's weight being split somehow between the columns the component is in. Similarly, each row's weight is related to the highest weighty specified for a component within that row. Extra space tends to go toward the rightmost column and bottom row.

The next section discusses constraints in depth, in the context of explaining how the example program works.

15.5 Additional Swings Components

Following table describes various additional swings components, their constructor definitions, methods and listener interfaces.

Table 15.1: Additional Swings Components

Component	**Common constructor parameters**	**Important methods**	**Useful Listeners**

JTabbedPane	• Tab placement	addTab()
		insertTab()
		getSelectedComponent()
		setSelectedComponent()
			ChangeListener

JScrollPane	• Component • Component, scrollbar options	setViewportView()	
JSplitPane	• Orientation, 2 components • For orientation, use JScrollPane.HORIZONTAL_SPLIT or JScrollPane.VERTICAL_SPLIT.	setDividerLocation()	
JComponent			MouseListener

The SwingX project exists to provide extensions to existing Swing components and Swing architecture (such as an ActionManager for helping manage Action instances). Whereas the JDNC project focuses on a general Application framework and solving a more specific set of application design problems, the SwingX project focuses exclusively on the raw components themselves. [...]

- JXDatePicker - Standard date chooser component
- JXPanel - Adds translucency to the standard JPanel
- JXGlassPanel - Dismiss on click & drop shadow functionality
- JXTitledPanel - Title bar added to the JXPanel ala JGoodies
- JXHyperLink - Extends JLabel, adds action listener support
- JXRadioGroupy, JXHyperLink - Extends JLabel, adds action listener support
- JXRadioGroup - Simplifies dealing with ButtonGroup & JRadioButton
- JXScrollUp - Extends JXTitledPanel, adds collapsing functionality
- JXStatusBar - Enhanced status bar functionality
- JXMonthView - Displays a month calendar
- JXImagePanel - Displays an image
- JXErrorPane - Enhanced standard error dialog

- JXEditorPane - Enhancements to JEditorPane
- JXList - Adds in place editing to JList
- JXTable - Adds filtering/sorting/highlighting/column hiding to JTable
- JXTree - Enhancements to JTree
- JXTreeTable - Combination of JTree & JTable [UI Specification]
- JXFindDialog - Standard find dialog
- JXComboBox - Enhanced JComboBox supporting embedded tables, etc
- JXButtonPanel - Smart panel for displaying buttons in OS specific & local specific order

15.6 Summary

In this unit we have discussed various Layout managers for positioning of controls. We have also discussed the power of java i.e. swings, hierarchy of swing package etc. At the end additional swings components like tabbed panes, scroll pane, split pane.

15.7 Self-Assessment Questions

1. Write Short note on-
   - A. FlowLayout Manager
   - B. Border Layout Manager
2. What is the difference between AWT and Swings.
3. Write a java applet program using swings methods and classes to find the factorial of a number which is accepted from keyboard. Use the recursive methods for factorial. Also write the HTML code for displaying the applet.

15.8 References

1. Programming with Java A Primer, E.Balaguruswamy Tata McGraw Hill Companies
2. Java Programming John P. Flynt Thomson 2nd
3. Java Programming Language Ken Arnold Pearson
4. The complete reference JAVA2, Herbert schildt. TMH
5. Big Java, Cay Horstmann 2nd edition, Wiley India Edition
6. Core Java, Dietel and Dietel
7. Java - Balaguruswamy
8. Java server programming, Ivan Bayross SPD