



S.Y.B.Sc.
(Computer Science)
SEMESTER - III (CBCS)

CORE JAVA

SUBJECT CODE : USCS302

Prof. Suhas Pednekar

Vice-Chancellor,
University of Mumbai,

Prof. Ravindra D. Kulkarni

Pro Vice-Chancellor,
University of Mumbai,

Prof. Prakash Mahanwar

Director,
IDOL, University of Mumbai,

Programme Co-ordinator : Shri Mandar Bhanushe

Head, Faculty of Science and Technology,
IDOL, University of Mumbai, Mumbai

Course Co-ordinator : Mr. Sumedh Shejole

Asst. Professor,
IDOL, University of Mumbai, Mumbai

Editor : Mr Milind Thorat

Assistant Professor
K J Somaiya Institute of Engineering & IT
Mumbai

Writers : Ahtesham Shaikh

Anjuman-i-Islam's Akbar Peerbhoy College
Vashi, Navi Mumbai

: Mrs. Vandana Maurya

B.K. Birla College(Autonomous), Kalyan

: Dr. Manisha Divate

Usha Pravin Gandhi College of Arts,
Science and Commerce, Mumbai

July 2022, Print - I

Published by

: Director
Institute of Distance and Open Learning ,
University of Mumbai,
Vidyanagari, Mumbai - 400 098.

DTP Composed and

: Mumbai University Press

Printed by

Vidyanagari, Santacruz (E), Mumbai - 400098

CONTENTS

Unit No.	Title	Page No.
Unit- I		
1.	The Java Language	01
2.	OOPS	18
3.	String Manipulations and Introduction to Packages	33
Unit - II		
4.	Exception Handling	50
5.	Multithreading	66
6.	I/O Streams	86
7.	Networking	102
Unit - III		
8.	Wrapper Classes	117
9.	Collection Framework	123
10.	Inner Classes	138
11.	AWT	146



Course: USCS302	TOPICS (Credits : 02 Lectures/Week:03) Core Java	
Objectives: The objective of this course is to teach the learner how to use Object Oriented paradigm to develop code and understand the concepts of Core Java and to cover-up with the pre-requisites of Core java.		
Expected Learning Outcomes: <div><div></div><div>1. Object oriented programming concepts using Java.</div><div>2. Knowledge of input, its processing and getting suitable output.</div><div>3. Understand, design, implement and evaluate classes and applets.</div><div>4. Knowledge and implementation of AWT package.</div></div>		
Unit I	The Java Language: Features of Java, Java programming format, Java Tokens, Java Statements, Java Data Types, Typecasting, Arrays OOPS: Introduction, Class, Object, Static Keywords, Constructors, this Key Word, Inheritance, super Key Word, Polymorphism (overloading and overriding), Abstraction, Encapsulation, Abstract Classes, Interfaces String Manipulations: String, String Buffer, String Tokenizer Packages: Introduction to predefined packages (java.lang, java.util, java.io, java.sql, java.swing), User Defined Packages, Access specifiers	15L
Unit II	Exception Handling: Introduction, Pre-Defined Exceptions, Try-Catch-Finally, Throws, throw, User Defined Exception examples Multithreading: Thread Creations, Thread Life Cycle, Life Cycle Methods, Synchronization, Wait() notify() notify all() methods I/O Streams: Introduction, Byte-oriented streams, Character- oriented streams, File, Random access File, Serialization Networking: Introduction, Socket, Server socket, Client –Server Communication	15L
	Wrapper Classes: Introduction, Byte, Short, Integer, Long, Float, Double, Character, Boolean classes Collection Framework: Introduction, util Package interfaces, List, Set, Map, List interface & its classes, Set interface & its classes, Map interface & its classes	

Unit III	Inner Classes: Introduction, Member inner class, Static inner class, Local inner class, Anonymous inner class AWT: Introduction, Components, Event-Delegation-Model, Listeners, Layouts, Individual components Label, Button, CheckBox, Radio Button, Choice, List, Menu, Text Field, Text Area	15L
Textbook(s): 1) Herbert Schildt, Java The Complete Reference, Ninth Edition, McGraw-Hill Education, 2014 Additional Reference(s): 1) E. Balagurusamy, Programming with Java, Tata McGraw-Hill Education India, 2014 2) Programming in JAVA, 2nd Ed, Sachin Malhotra & Saurabh Choudhary, Oxford Press 3) The Java Tutorials: http://docs.oracle.com/javase/tutorial/		

THE JAVA LANGUAGE

Unit Structure

- 1.0 Objectives
- 1.1 Features of Java
- 1.2 Java programming format
- 1.3 Summary
- 1.4 Textbook
- 1.5 Additional References
- 1.6 Questions

1.0 OBJECTIVES:

The objective of this chapter is to learn the basic building blocks of java and understand the concepts of Core Java and to cover-up with the pre-requisites of Core java, Advanced Java, J2EE and J2ME.

Topics:

Features of Java, Java programming format, Java Tokens, Java Statements, Java Data Types, Typecasting, Arrays

1.1 FEATURES OF JAVA

- **Simple:** A very simple, easy to learn and understand language for programmers who are already familiar with OOP concepts. Java's programming style and structure follows the lineage of C, C++ and other similar languages makes the use of java efficiently.
- **Object-oriented:** Java is object oriented. Java inherits features of C++. OOP features of java are influenced by C++. OOP concept forms the heart of java language that helps java program in survive the inevitable changes accompanying software development.
- **Secure, Portable and Robust:** Java programs are safe and secure to download from internet. At the core of the problem is the fact that malicious code can cause its damage due to unauthorized access gained to system resources. Java achieved this protection by confining a program to the Java execution environment and not allowing it access to other parts of the computer. The *same* code must work on *all* computers. Therefore, some means of generating portable executable code was needed. The multi-platform environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of

systems. Thus, the ability to create robust programs was given a high priority in the design of Java. To gain reliability, Java restricts a few key areas and forces to find your mistakes early in program development. At the same time, Java frees a programmer from having to worry about many of the most common causes of programming errors.

➤ **Multithreaded:** Java supports multithreaded programming, which allows a programmer to write programs that performs multiple tasks simultaneously. The Java run-time system comes with an elegant and sophisticated solution for multi-process synchronization that helps to construct smoothly running interactive systems.

➤ **Architecture-neutral:** Java was designed to support applications on networks composed of a variety of systems with a variety of CPU and operating system architectures. With Java, the same version of the application runs on all platforms. The Java compiler does this by generating bytecode instructions which have nothing to do with particular processor architecture. Rather, they are designed to be both easy to interpret on any machine and easily translated into native machine code on the fly.

➤ **Interpreted & High performance:** Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be executed on any system that implements the Java Virtual Machine. Most previous attempts at cross-platform solutions have done so at the expense of performance. The Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code.

➤ **Distributed:** Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. Java also supports *Remote Method Invocation (RMI)*. This feature enables a program to invoke methods across a network.

➤ **Dynamic:** Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the Java environment, in which small fragments of byte code may be dynamically updated on a running system.

1.2 JAVA PROGRAMMING FORMAT

1) Package Section	It must be the first line of a java program or can be omitted if the class is to be kept only in the default package. The package statement defines a namespace in which classes are stored, based on functionality. If omitted, the classes are put into the default package, which has no name.
2) Import Section	Specifies the location to use a class or package into a program.
3) Class / Interface section	A java program may contain several classes or interfaces.
4) Class with Main Method	Every Java stand-alone program requires the main method as the starting point of the program. This is an essential part of a Java program. There may be many classes in a Java program code file, and only one class defines the main method.

Example:

// ---- 1 **Package Section** -----

Package mypack;

// ----- 2 **Import Section** -----

import java.util.Date; // This line will import only one class from the util package

import java.awt.*; // This line will import all classes available in awt package

// ----- 3 **Class / Interface Section** -----

class A {

 // Class Body

}

interface B {

 // Interface Body

}

// ----- 4 **Main Method Section** -----

public class Test{

 public static void main(String[] args){

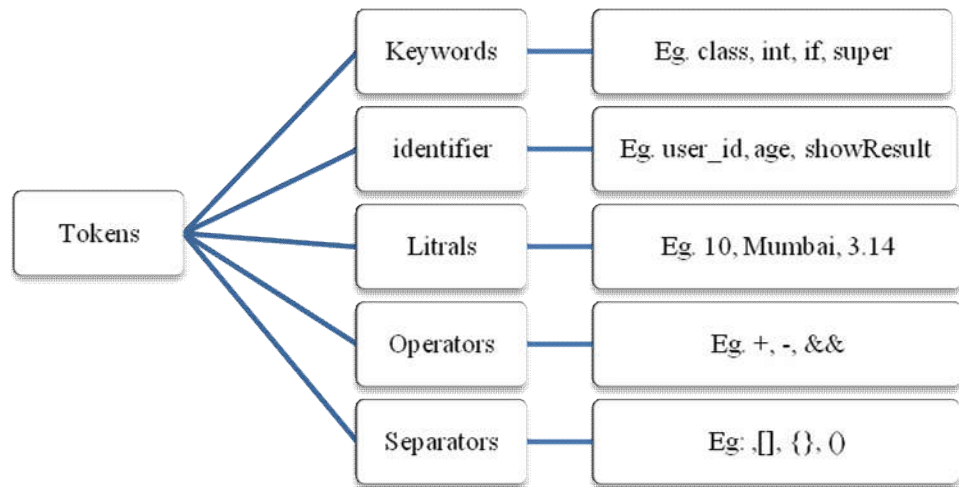
 // body of main method

 }

}

2 Java Tokens

Tokens are the basic building blocks of the java programming language that are used in constructing expressions, statements and blocks. The different types of tokens in java are:



1. Keywords: these words are already been defined by the language and have a predefined meaning and use. Key words cannot be used as a variable, method, class or interface etc.

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

2. Identifiers: Identifiers are used to name a variable, method, block, class or interface etc. Java is case-sensitive. Identifier may be any sequence of uppercase and lowercase letters, numbers, or the underscore characters.

Rules for defining identifier:

- All variable names must begin with a letter of the alphabet. The dollar sign and the underscore are used in special case.
- After the first initial letter, variable names may also contain letters and the digits 0 to 9. No spaces or special characters are allowed.
- The name can be of any length.
- Uppercase characters are distinct from lowercase characters. Variable names are case-sensitive.

- Java keyword (reserved word) cannot be used for a variable name.

Examples of valid identifiers in java: num1, name, Resi_addr, Type_of_road, Int etc.

Examples of invalid identifiers in java: 1num, full-name, Resiaddr, Type*ofroad, int etc.

3. Literals: Literals are the value assigned to a variable;

Example: 10, "Mumbai", 3.14, 'Y', '\n' etc.

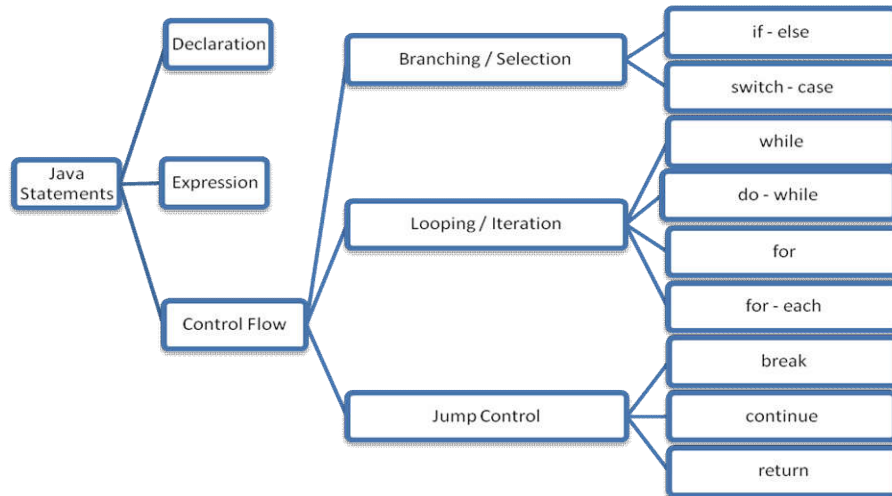
4. Operators: Operators are the symbols that performs operations. Java contains different types of operators like Arithmetic Operator (+, -, *, /, %), Logical Operator (&&, ||, ~,), Relational Operator(<, <=, >, >=, !=, ==), Bitwise Operators (&, |, ~), Shift Operators (<<, >>, >>>) , Assignment Operators(=, +=, -=, *=, /=, %=), Conditional Operator (?), InstanceOf Operator

5. Separators: Separators are used to separate words, expressions, sentences, blocks etc.

Symbol	Name	Description
	Space	Used to separate tokens.
;	Semicolon	Used to separate the statements
()	Parentheses	Used to contain the lists of parameters in method definition and invocation. Also used for defining the precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contains the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[]	Brackets	Used to declare array types. Also used when dereferencing array values.
,	Comma	Separates consecutive identifiers in a variable declarations. Also used to chain statements together inside a for statement
.	Period	Used to separate packages names from subpackages and classes. Also used to separate a variable or method from a reference variable.

3 Java Statements

A statement specifies an action in a Java program. Statements in Java can be broadly classified into three categories:



1. Declaration

A declaration statement is used to declare a variable, method or class.

For example: `int num;`

`double PI = 3.14;`

`String name="University of Mumbai;`

`int showResult(int a, int b);`

2. Expression

An *expression* is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language, that evaluates to a single value.

Arithmetic Expression: $(A + B) * C - (D \% E) * (-F + G)$

Logical Expression: $\sim(m > n \ \&\& \ x < y) \ != \ (m \leq n \ || \ x \geq y)$

3. Flow Control

By default, all the statements in a java program are executed in the order they appear in the program code. However sometime a set of statements need to be executed and a part to be skipped, also some part need to be repeated as long as a condition is true or till some fix number of iteration. Java programming language uses flow control statements to cause the flow of execution to advance and branch based on changes to the state of a program. Java's program control statements can be put into the following categories: Branching / selection, Looping / iteration, and jump control. Selection statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable. Iteration statements enable program execution to repeat one or more statements (that is, iteration statements form loops). Jump statements allow your program to execute in a nonlinear fashion.

I. Selection / Branching:

Java supports two selection statements: `if` and `switch`. These statements allow you to control the flow of your program's execution based upon conditions known only during run time.

➤ **if**

The `if` statement is Java's conditional branch statement. It can be used to route program execution through two different paths.

General Syntax:

```
if(condition)
statement1 / { if Block };
else
statement2 / { else Block};
```

Here, each statement may be a single statement or a compound statement enclosed in curly braces (that is, a block). The condition is any expression that returns a boolean value. The `else` clause is optional.

Example:

```
class ifelsetest{
public static void main(String[] args){
int num=10;
if(num%2 == 0){
System.out.println("Number is EVEN");
}

    else {

        System.out.println("Number is ODD");
    }

}

}

}
```

➤ **The if-else-if Ladder**

A common programming construct that is based upon a sequence of nested ifs is the `if-else-if` ladder. General Syntax:

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

The `if` statements are executed from the top down. As soon as one of the conditions controlling the `if` is true, the statement associated with that `if` is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final `else` statement will be executed. The final `else` acts as

a default condition; that is, if all other conditional tests fail, then the last else statement is performed. If there is no final else and all other conditions are false, then no action will take place.

Example:

```
class ladderifelseetest{
public static void main(String[] args){
int num=10;
if(num> 0){
System.out.println("Number is +VE");
}

    else if(num<0){

        System.out.println("Number is -VE");

    }

else {

        System.out.println("Number is ODD");

    }

}

}
}
```

➤ **switch**

The switch statement is Java's multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of if-else-if statements.

General Syntax:

```
switch (expression)
{
case value1:
// statement sequence
break;
case value2:
// statement sequence
break;
.
.
}
```

```

casevalueN :

//statement sequence break;

default:

// default statement sequence

}

```

For versions of Java prior to JDK 7, expression must be of type byte, short, int, char, or an enumeration. JDK 7 onwards the switch expression can also be of type String.

Example:

```

class switchcasetest{
public static void main(String[] args){
int num=10;
switch(num){
case 1:
System.out.println("Monday");

                break;

case 2:
System.out.println("Tuesday");

                break;

case 3:
System.out.println("Wednesday");

                break;

case 4:
System.out.println("Thursday");

                break;

case 5:
System.out.println("Friday");

                break;

case 6:
System.out.println("Saturday");

                break;

case 7:
System.out.println("Sunday");

                break;

```

```

default:
System.out.println("~~~ Invalid Week day No ~~~");

        break;

    }
}
}

```

II. Branching / Iteration Statements

Java's iteration statements are for, while, and do-while. These statements create what we commonly call loops. As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met. As you will see, Java has a loop to fit any programming need.

➤ while

The while loop is Java's most fundamental loop statement. It repeats a statement or block while its controlling expression is true.

General Syntax:

```

while(condition)
{
    // body of loop
}

```

The condition in java must be strictly a Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When condition becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

Examples:

```

public class WhileDemo
{
    public static void main(String args[])
    {
        int n = 10;
        while(n > 0)
        {
            System.out.println("Count Doun Value" + n);
            n - -;
        }
    }
}

```

```

}
}
}

```

➤ **do-while**

Sometimes it is desirable to execute the body of a loop at least once, even if the conditional expression is false to begin with. In other words, there are times when you would like to test the termination expression at the end of the loop rather than at the beginning. Java provides a loop that does just that: the do-while. The do-while loop always executes its body at least once, because its conditional expression is at the bottom of the loop.

General Syntax:

```

do {
// body of loop
} while (condition);

```

Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates. As with all of Java's loops, condition must be a Boolean expression.

Examples:

```

public class DoWhileDemo
{
public static void main(String args[])
{
    int n = 10;
do{
System.out.println("Loop Executed Once even if condition is False" );
n - -;
} while(n > 10) ;
}
}

```

➤ **For**

The for loop operates as follows. When the loop first starts, the initialization portion of the loop is executed. Generally, this is an expression that sets the value of the loop control variable, which acts as a counter that controls the loop. It is important to understand that the

initialization expression is executed only once. Next, condition is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates. Next, the iteration portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

General Syntax:

```

        for(initialization; condition; increment / decrement expression)
    {
//      loop body
    }

```

Examples:

```

public class ForDemo
{
    public static void main(String args[])
    {
        for(int x = 1 ; x <=10 ; x++)
        {
            System.out.println("Loop Variable Value is : " + x);
        }
    }
}

```

There will be times when you will want to include more than one statement in the initialization and iteration portions of the for loop.

Example:

```

public class TwoVarFor
{
    public static void main(String args[]) {
        int a, b ;
        for(a=1, b=4; a<= b; a++, b--){
            System.out.println("value of A is : "+a+"    Value of B is : "+b);
        }
    }
}

```

➤ The For-Each Loop

A for-each loop by using the keyword `for-each`, Java adds the for-each capability by enhancing the `for` statement. The advantage of this approach is that no new keyword is required, and no pre-existing code is broken. The for-each style of `for` is also referred to as the enhanced for loop.

General Syntax

```
for( type itr-var : collection)
{
//    statement-block;
}
```

Example:

```
public class ForEachDemo {
public static void main(String args[])
{ String names[] = {"Mumbai", "Pune", "Nagpur", "Aurangabad",
"Thane", "Nasik" };
for (String x : names)
{ System.out.println("Name of the City in Maharashtra is: " + x);
}
System.out.println("~~~~~Printing on City Names Done~~~~~"); }
}
```

III. Jump Control Statement

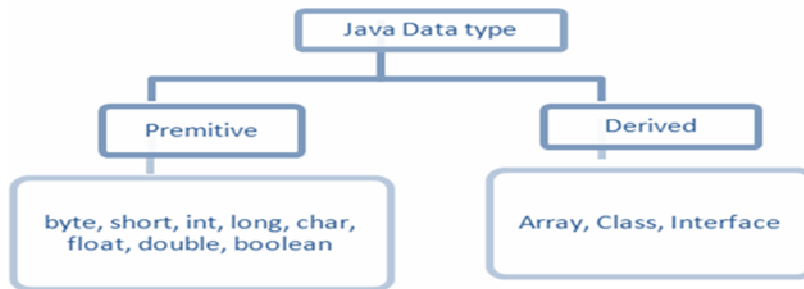
Java supports three jump statements: `break`, `continue`, and `return`. These statements transfer control to another part of a java program.

In Java, the `break` statement has three uses. First, as you have seen, it terminates a statement sequence in a `switch` statement. Second, it can be used to exit a loop. Third, it can be used as a form of `goto` statement in C/C++.

Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a `goto` just past the body of the loop, to the loop's end. The `continue` statement performs such an action. In `while` and `do-while` loops, a `continue` statement causes control to be transferred directly to the conditional expression that controls the loop. In a `for` loop, control goes first to the iteration portion of the `for` statement and then to the conditional expression. For all three loops, any intermediate code is bypassed.

The last control statement is `return`. The `return` statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.

4 Java Data Types



The Primitive Types: Java defines eight *primitive* types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **boolean**. The primitive types are also commonly referred to as *simple* types.

These can be put in four groups:

1. **Integers** This group includes **byte**, **short**, **int**, and **long**, which are for whole-valued signed numbers.

Name		Width	Range
byte	8	1 byte	−128 to 127
short	16	2 byte	−32,768 to 32,767
int	32	4 byte	−2,147,483,648 to 2,147,483,647
long	64	6 byte	−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

The smallest integer type is **byte**. Variables of type **byte** are especially useful when you're working with a stream of data from a network or file. They are also useful when you're working with raw binary data that may not be directly compatible with Java's other built-in types. The most commonly used integer type is **int**. **long** is a signed 64-bit type and is useful for those occasions where an **int** type is not large enough to hold the desired value.

2. **Floating-point numbers** This group includes **float** and **double**, which represent numbers with fractional precision.

Name		Width	Approximate Range
float	32	4 byte	1.4e−045 to 3.4e+038
double	64	8 byte	4.9e−324 to 1.8e+308

The type **float** specifies a *single-precision* value that uses 32 bits of storage. Single precision is faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small. Variables of type **float** are useful when you need a fractional component, but don't require a large degree of precision. Double precision, as denoted by the **double** keyword, uses 64 bits to store a value. Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations.

3. Characters This group includes **char**, which represents symbols in a character set, like letters and numbers. C++ char is 8 bit i.e. 256 symbols while java char uses 16 bit i.e. 65536 symbols to represent *Unicode* characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages and is a unification of character sets, such as Latin, Greek, Arabic, Cyrillic Hebrew, Katakana, Hangul, and many more. There is no signed char in java. char can also be used as an integer type on which you can perform arithmetic operations.

Example: if char ans = 'A' then ans = ans + 5 will result 'F'.

4. Boolean This group includes **boolean**, which is a special type for representing true/false values.

5 Typecasting

As a part of Java's safety and robustness Java is a strongly typed language. Every variable has a type, every expression has a type, all assignments, whether explicit or via parameter passing in method calls is checked for type compatibility and every type is strictly defined. There are no automatic coercions or conversions of conflicting types. If the two types are compatible and the target type is equal to or larger than the source type JVM performs automatic type conversion. This type of conversion is also called implicit conversion or automatic type casting or widening conversion.

byte → short → int → long or int → float → double

If the two types are incompatible and the target type is smaller than the source type then typecasting is required conversion. This type of conversion is also called explicit conversion or manual type casting or narrowing conversion.

General syntax: (type_type) value|expression

Eg. int a = (int) 3.14;

Truncation will occur when a floating-point value is assigned to an integer type, because integers do not have fractional component. For example, if the value 3.14 is assigned to an integer, the resulting value will simply be 3; the 0.14 will have been truncated.

Java also performs type promotion while evaluating mix mode expression. Java defines several *type promotion* rules that apply to expressions. They are as follows: First, all **byte**, **short**, and **char** values are promoted to **int**, as just described. Then, if one operand is a **long**, the whole expression is promoted to **long**. If one operand is a **float**, the entire expression is promoted to **float**. If any of the operands are **double**, the result is **double**.



6 Arrays

An array is a collection of similar data type, identified by a common name and stored in consecutive memory location. Array elements can be conveniently accessed using index number. Java arrays are reference type. A *one-dimensional array* is a list of like typed variables. The general form of a one-dimensional array declaration is

[access_specifier] type var-name[];

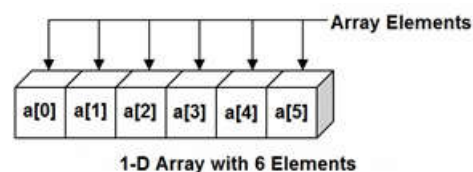
or *[access_specifier] type[] var-name;*

Here, *type* declares the element type (also called the base type) of the array. The element type determines the data type of each element that comprises the array. Thus, the element type for the array determines what type of data the array will hold. For example, the following declares an array named **temp_jan** with the type “array of int”: `public float temp_jan[]`

The declaration creates a reference to an array. The fact is that actual array does not exist in memory. Memory allocation is done using “new” operator.

```
temp_jan = new float[31]
```

This will create 31 float variable in memory and assign the base reference to temp_jan.



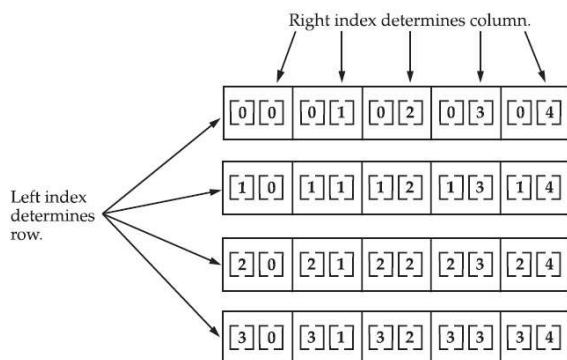
Declaration and initialization of array can be combined eg. `public float temp_jan[] = new float[31];`

two-dimensional array is a list of list of like typed variables or an array of array. The general form of a one-dimensional array declaration is

[access_specifier] type var-name[][];

Eg. `public int m1[][] = new int[3][3]`

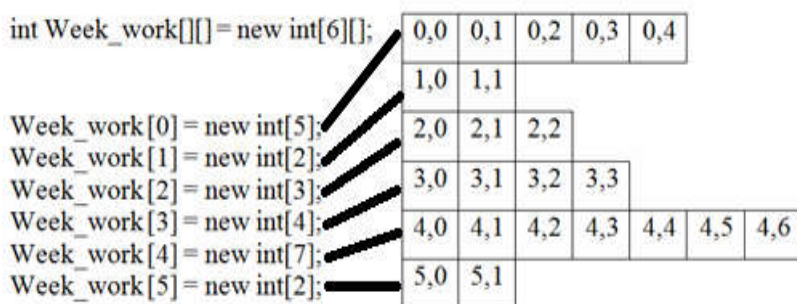
This will declare a 2D array of int to hold 3 rows and 3 columns.



Given: `int twoD [] [] = new int [4] [5];`

A conceptual view of a 4 by 5, two-dimensional array

Java allows creating a multi-dimensional array to be created by declaring the first dimension and allocate the remaining dimension separately. This creates a jagged array or variable size array with different rows having different number of columns.



1.3 SUMMARY:

The chapter helps to learn the features of java language, the format of java program, basic building blocks of java and understand the concepts of Core Java and to cover-up with the pre-requisites of Core java, Advanced Java, J2EE and J2ME.

1.4 TEXTBOOKS:

- 1) Herbert Schildt, Java The Complete Reference, Ninth Edition, McGraw-Hill Education, 2014

1.5 ADDITIONAL REFERENCES:

- 1) E. Balagurusamy, Programming with Java, Tata McGraw-Hill Education India, 2014
- 2) Programming in JAVA, 2nd Ed, Sachin Malhotra &SaurabhChoudhary, Oxford Press

1.6 QUESTIONS:

1. Explain the feature of Java
2. Explain the For-Each Loop with example?



OOPS

Unit Structure

- 2.0 Objectives
- 2.1 Introduction
- 2.2 Class
- 2.3 Object
- 2.4 Static Keywords
- 2.5 Constructors
- 2.6 this Key Word
- 2.7 Inheritance
- 2.8 super Keyword
- 2.9 Polymorphism (overloading and overriding)
- 2.10 Abstraction
- 2.11 Encapsulation
- 2.12 Abstract Classes
- 2.13 Interfaces
- 2.14 Summary
- 2.15 Textbook
- 2.16 Additional References
- 2.17 Questions

2.0 OBJECTIVES

The objective of this chapter is to learn the basic concepts of Object Oriented Programming and its implementation in java to develop the code to cover-up with the pre-requisites of Core java, Advanced Java, J2EE and J2ME.

Topics:

2.1 INTRODUCTION

Object oriented programming implements object oriented model in software development. OOP is based on three principles i.e. Encapsulation, Inheritance and polymorphism. OOP allows decomposing a large system into small object.

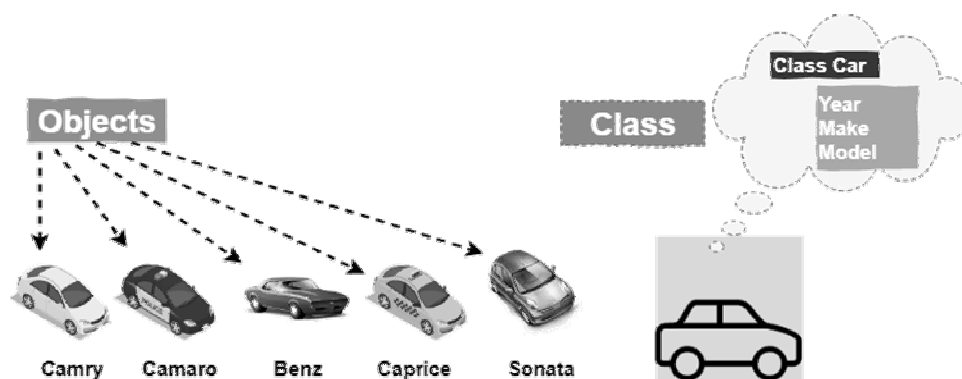
Encapsulation is the mechanism of binding together code and the data it manipulates, and keeps both safe from outside interference and misuse. It is like a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface.

Inheritance is the process by which one object acquires the properties of another object. It is a way of making new classes using existing one and redefining them.

Polymorphism (Greek meaning “many forms”) is a feature that allows one interface to be used for a general class of actions. More generally, the concept of polymorphism is often expressed by the phrase “one interface, multiple methods.” This means that it is possible to design a generic interface to a group of related activities. This helps reduce complexity by allowing the same interface to be used to specify a general class of action.

2.2 CLASS

A class is a blue print for creating objects. A class is a group of objects which have common properties. A class defines the data and code that can be shared by a set of objects. Each object of a given class contains the structure and behavior defined by the class, as if it were stamped out by a mold in the shape of the class.



```
[access_specifier] [modifier] class <class_name>{
```

Fields

Methods

Constructors

Blocks

Nested class and interface

```
}
```


The data, or variables, defined within a **class** are called instance variables. The code is contained within methods. The methods, constants and variables etc. defined within a class are called members of the class.

2.3 OBJECT

An entity that has state and behavior is known as an object. An object has three characteristics:

- **State:** represents the data or value of an object.
- **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
- **Identity:** An object identity is a unique ID.

An object represents a class during program execution. Thus, a class is a template for an object, and an object is an instance of a class. Because an object is an instance of a class, you will often see the two words object and instance used interchangeably.

There are five different ways to create objects in java:
Using new keyword:

```
Complex com = new Complex(10, 20);
```

1. Using Class.forName():

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

2. Using clone():

```
Complex com1 = com.clone();
```

3. Using Object Deserialization

```
ObjectInputStream ois = new ObjectInputStream(some data);
```

```
MyObject object = (MyObject) ois.readObject();
```

Using newInstance() method

```
Object obj =
```

```
DemoClass.class.getClassLoader().loadClass("DemoClass").new  
Instance ();
```

2.4 STATIC KEYWORDS

Static is a non-access modifier in Java, it is used with variables, methods, blocks and nested class. It is a keyword that are used for share the same variable or method of a given class. This is used for a constant variable or a method that is the same for every object of a class. The main method of a class is generally labeled static. The static keyword is used in java mainly for memory management. No object needs to be created to use static variable or call static methods, just put the class name before the static variable or method to use them. Static method cannot call non-static method. The static variable allocate memory only once in class area at the time of class loading. It is use to make our program memory efficient.

When a variable is declared as static, then a single copy of variable is created and shared among all objects at class level. Static variables are, essentially, global variables. All instances of the class share the same static variable and can be created at class-level only.

When a method is declared with static keyword, it is known as static method. The most common example of a static method is `main()` method. Any static member can be accessed before any objects of its class are created, and without reference to any object. Methods declared as static can only directly call other static methods and can only directly access static data. They cannot refer to `this` or `super`.

2.5 CONSTRUCTORS

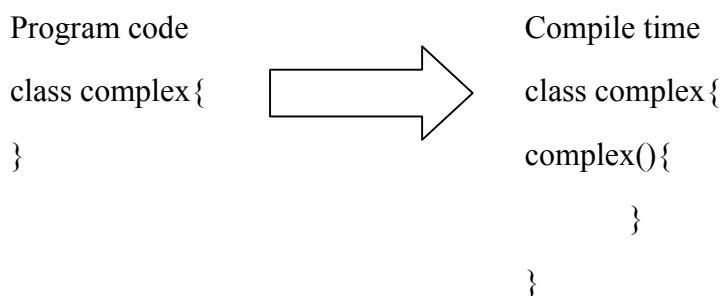
A constructor initializes an object at the time of creation. It has the same name as the class in which it resides. The constructor is automatically called when the object is created, before the `new` operator completes. Constructors have no return type, not even `void`. This is because the implicit return type of a class' constructor is the class type itself. It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

There are three rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

Default constructor: If not implemented any constructor by the programmer in a class, Java compiler inserts a default constructor with empty body into the code, this constructor is known as default constructor. If user defines any parameterized constructor, then compiler will not create default constructor and vice versa if user don't define any constructor, the compiler creates the default constructor by default during compilation

Eg:



no-arg constructor: Constructor with no arguments is known as no-arg constructor. The signature is same as default constructor; however body can have any code unlike default constructor where the body of the constructor is empty.

Eg: class complex{

int real, img;

public complex (){

real=0;

img=0;

}

}

Parameterized constructor: Constructor with arguments is known as no-arg constructor. The signature is same as default constructor; however body can have any code unlike default constructor where the body of the constructor is empty.

class complex {

inta,b,c;

public complex (int r, inti){

real=r;

img=i;

}

}

Copy Constructor: Values from one object to another object can be copied using constructor or by clone() method of Object class.

A copy constructor is a parameterized constructor but the parameter is the reference of containing class.

class complex {

inta,b,c;

public complex (complex c){

real=c.real; img=c.img;

}

}

Java Constructor	Java Method
A constructor is used to initialize the state of an object.	A method is used to expose the behavior of an object.
A constructor must not have a return type.	A method must have a return type.
The constructor is invoked implicitly.	The method is invoked explicitly.
The Java compiler provides a default constructor if you don't have any constructor in a class.	The method is not provided by the compiler in any case.
The constructor name must be same as the class name.	The method name may or may not be same as class name.

Constructor Overloading: Sometimes there is a need of initializing an object in different ways. This can be done using constructor overloading. E.g. A frame object can be created using default constructor or using title of the frame. Multiple constructors can be created by changing the no. of parameters, type of parameters, order of parameters or combination of any.

Eg.

```
class Complex{
    intrel, img;
    doublerell, imgg
    Complex(){}
    Complex(int r, inti){ rel=r;  img=i;    }
    Complex(double r, double i){rell=r;    imgg=i;    }
    Complex (int r, double r){rel = r;  imgg=i;    }
    Complex(double r, inti){rell=r;  img=i;          }
    Complex(Complex c){rel=c.rel;  img=c.img;}
}
```

2.6 THIS KEY WORD

“this” is a reference to object itself. **‘this’ keyword can be used to refer current class instance variables, to invoke current class constructor, to return the current class instance, as method parameter, to invoke current class method and as an argument in the constructor call.** “*this*” keyword when used in a constructor can only be the first statement in Constructor and constructor can have either *this* or *super* keyword but not both.

```
class A{
    int a=10;

    public void show(){
        double a=100.200;

        System.out.println("Value of A is : "+a);
    }
}
```

The above program code will display “Value of A is : 100.200”, because the preference will always go to local variable or the variable with immediate scope.

```
System.out.println("Value of A is : "+this.a);
```

The above statement will display “Value of A is: 10”, because the reference “this” will point to current instance variable “a” of the class.

```
class A{
    A(){
        this(111);

        System.out.println(" Default Constructor Called .....");
    }

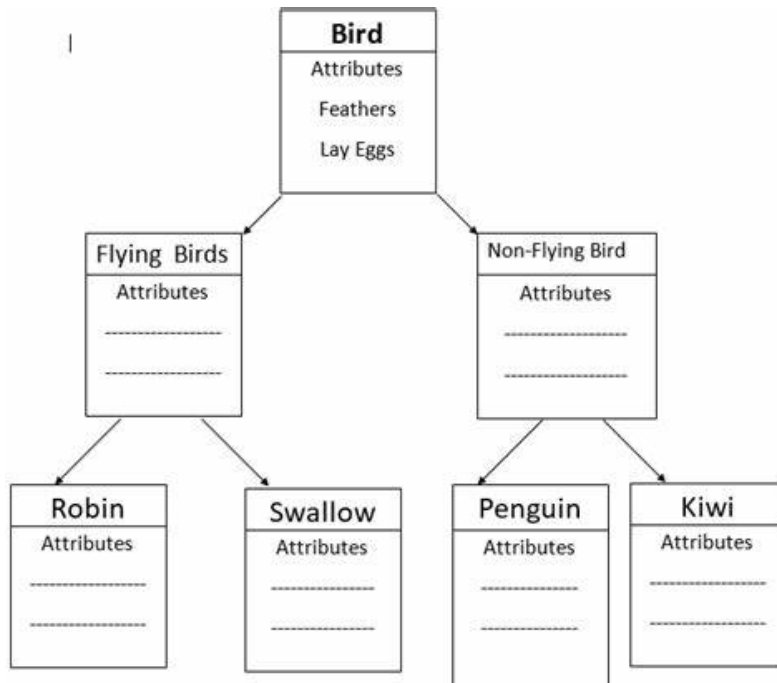
    A(int a){
        System.out.println(" Parameterized Constructor Called");
    }
}
```

The call `A obj = new A()` will create an object of “A” using default constructor and will also call the parameterized constructor by passing value 111 using “this”.

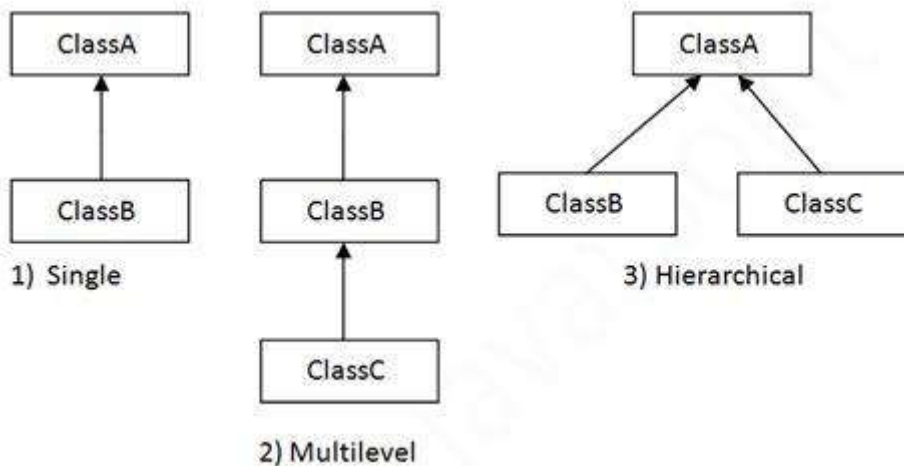
2.7 INHERITANCE

OOPS

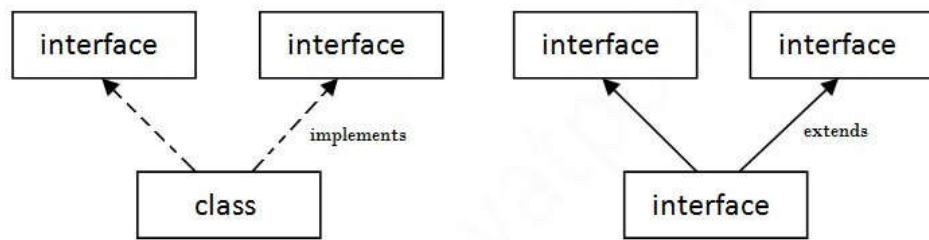
Inheritance is an important concept of OOP. It is the mechanism in java by which one class is allows inheriting the fields and methods of another class. Inheritance facilitates code reusability to reuse the fields and methods of the existing class. The class from which a new class is created is called as a parent, base or super class and the new class is also called as child, derived or sub class.



There are Three types of inheritance in Java using classes:



Java supports multiple inheritance using Interfaces:



Multiple Inheritance in Java

Single Inheritance: In Single Inheritance one class extends another class (one class only).

```

class A { double pi=3.14;
public void add(int a , int b){
System.out.println("Add = "+(a+b));} }
class B extends A {
int Max=100;
public void sub(int a, int b){
System.out.println("Sub = "+(a-b));} }

```

If an object obj is created for class B, it will also have methods and properties from its parent class.

Multilevel Inheritance: In Multilevel Inheritance, one class can inherit from a derived class. Hence, the derived class becomes the base class for the new class.

```

class A { int a=10;}
class B extends A {int b=20;}
class C extends B {int c=30;}

```

If an object obj is created for class C, it will also have methods and properties from A & B parent classes.

Hierarchical Inheritance: In Hierarchical Inheritance, one class is inherited by many sub classes.

```

class Person { String name, gender;          int age;}
class Student extends Person { int roll;      }
class Teacher extends Person { intempid;    }
class Doctor extends Person { String specialty; }

```

In the above example Person class properties will be inherited in Student, teacher and Doctor.

2.8 SUPER KEY WORD

“super” is the reference to the parent class. Super keyword can be used to access parent class variable, method and **to invoke parent class constructor**.

```

class A{
String var="ANAAS"; }
class B extends A{
String var="AARISH";}
public void show(){
String var="NASHRAH";
System.out.println("Variable value is :"+var);
System.out.println("Variable value is :"+this.var);
System.out.println("Variable value is :"+super.var);
}}

```

In the above example var refer to the local context, this refer to the class variable and super will refer to super class member.

- **super** keyword can only be the first statement in Constructor.
- A constructor can have either **this** or **super** keyword but not both.

```

class A{
A(){System.out.println("Hello at A");}
A(String name){System.out.println(name+ " Hello at A");}}
class B extends A{
B(String name){System.out.println(name+ " Hello at B");}}
}

```

B obj = new B("AnAriNash") will create an object of B using string argument and will create an object of class A using default (non-parametrised) constructor.

```

class A{
A(){System.out.println("Hello at A");}
A(String name){System.out.println(name+ " Hello at A");}}
class B extends A{
B(String name){
super(name);
System.out.println(name+ " Hello at B");}}
}

```

The above program will invoke the parameterized constructor of class A.

2.9 POLYMORPHISM (OVERLOADING AND OVERRIDING)

Method Overloading is a mechanism in which a class allows more than one method with same name but with different prototype. Multiple methods can be created by changing the no. of parameters, type of parameters, order of parameters or combination of any. The method binding is done by the compiler at compile time and fix the calling method based on the actual parameter matching or by using implicit type conversion. This is also called as static binding, early binding or compile time polymorphism.

```
Class MyMath{

public void add(){System.out.println("Addition of 10 and 20 is
"+(10+20));}

public void add (int n1, int n2){ System.out.println(n1+n2);    }

public void add (double n1, double n2){ System.out.println(n1+n2);  }

public void add (int n1, double n2){ System.out.println(n1+n2); }

public void add (double n1, int n2){ System.out.println(n1+n2); }

public void add (Complex n1, Complex n2){ System.out.println(n1+n2);
    }}

```

Method Overriding is a mechanism in which a method in a child class that is already defined in the parent class with the same method signature — same name, arguments, and return type. Method overriding is used to provide the specific implementation of a method which is already provided by its superclass. The method binding is done by the java interpreter at run time and fix the calling method based on the latest implementation in class hierarchy. This is also called as dynamic binding, late binding or run time polymorphism.

```
class MyClass1 {

public void add(int a, int b){ return a+b;}

}

class MyClass2 extends MyClass1 {

@Override

public void add(int a, int b){ return 5*a+50*b; }

}

```

@override annotation tells the compiler that the method is meant to **override** a method declared in a superclass.

	Overloading	Overriding
1	More than one method with same name but different signature in same scope.	More than one method with same name and same signature in different scope.
2	Parameters are different	Parameters are same
3	Binding at compile time	Binding at run time.
4	Method return type may or may not be same	Method return type should be same.
5	Allowed for static method	Not allowed for static method
6	Cannot be prevented	Can be prevented by declaring a method as static or final.
7	Occurs in same class.	Occurs in sub class.

2.10 ABSTRACTION

Abstraction is a process of hiding the implementation details and showing only functionality to the user. Abstraction is selecting data from a larger pool to show only the relevant details to the object. It helps to reduce programming complexity and effort. In Java, abstraction is accomplished using Abstract classes and interfaces. Abstraction can be achieved using Abstract Class and Abstract Method in Java.

2.11 ENCAPSULATION

Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code. The class is Java's smallest unit of abstraction. Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:

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

The three access modifiers, **private**, **public**, and **protected**, provide a variety of ways to produce the many levels of access required by these categories.

2.12 ABSTRACT CLASSES

A class which is declared "abstract" is called as an abstract class. It can have abstract methods as well as concrete methods. A normal class cannot have abstract methods. Abstract classes help to describe generic types of behaviors and object-oriented programming class hierarchy. It also

describes subclasses to offer implementation details of the abstract class. Abstract class cannot be instantiated and is only used through inheritance. A “final” keyword cannot be used with abstract class.

Abstract Method: A method without a body is known as an Abstract Method. It must be declared in an abstract class. The abstract method will never be final because the abstract class must implement all the abstract methods. Abstract methods do not have an implementation; it only has method signature

If a class is using an abstract method they must be declared abstract. The opposite cannot be true. This means that an abstract class does not necessarily have an abstract method. If a regular class extends an abstract class, then that class must implement all the abstract methods of the abstract parent otherwise this class will also become abstract. Abstract methods are mostly declared where two or more subclasses are also doing the same thing in different ways through different implementations.

```
abstract class MyMath1 {

    public int add(int a, int b){return a+b;}

    public abstract int sub(int a, int b);

}

class MyMath2 extends MyMath1 {

    @Override

    Public int sub(int a, int b){ return a-b;}

    Public int mul(int a, int b){ return a*b;}

    Public int div(int a, int b){return a/b;}

}
```

2.13 INTERFACES

An interface is like a class but, it has static constants and abstract methods only. An **interface in java** is a blueprint of a class. The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body.

An interface is declared using **interface** keyword. It is used to provide total abstraction. That means all the methods in interface are declared with empty body and are public and all fields are public, static and final by default. A class that implement interface must implement all the methods declared in the interface. To implement interface use **implements** keyword.

Interface can extend another interface. Java allows multiple inheritance using interface.

```

interface<interface_name> [extends [Interface_name]]{

// fields

// Methods

}

```

OOPS

Example:

```

Interface MyMath{

public static final double PI=3.14;

public int add(int a, int b);

public int sub(int a, int b);

public int mul(int a, int b);

public int div(int a, int b);

}

```

Difference between Abstract Class and Interface

Abstract Class	Interface
An abstract class can have both abstract and non-abstract methods.	The interface can have only abstract methods.
It does not support multiple inheritances.	It supports multiple inheritances.
It can provide the implementation of the interface.	It cannot provide the implementation of the abstract class.
An abstract class can have protected and abstract public methods.	An interface can have only have public abstract methods.
An abstract class can have final, static, or static final variable with any access specifier.	The interface can only have a public static final variable.

2.14 SUMMARY:

The chapter helps to learn the concept of OOPS, like Classes, Inheritance, the keywords associated with OOP, implementation of polymorphism and Abstraction in java.

2.15 TEXTBOOK(S):

- 1) Herbert Schildt, Java The Complete Reference, Ninth Edition, McGraw-Hill Education, 2014

2.16 ADDITIONAL REFERENCE(S):

- 1) E. Balagurusamy, Programming with Java, Tata McGraw-Hill Education India, 2014
- 2) Programming in JAVA, 2nd Ed, Sachin Malhotra & Saurabh Choudhary, Oxford Press

2.17 QUESTIONS:

1. What is interface? Explain with example?
2. Write a short note on abstract class?
3. Explain the difference between Overloading & Overriding.
4. Explain the Difference between constructor and method in Java.



STRING MANIPULATIONS AND INTRODUCTION TO PACKAGES

Unit Structure

3.0 Objectives

3.1 String Manipulations

3.2 Packages

3.3 Summary.

3.4 Textbook

3.5 Additional Reference(s)

3.6 Questions:

3.0 OBJECTIVES:

The objective of this chapter is to learn the classes used in string manipulation. The different methods associated with string manipulation. String is a commonly used in many desktop and web application and has a wide range of applications. The chapter further introduces the concept of Package and access specifiers.

3.1.1 Topics: String, StringBuffer, String Tokenizer & packages.

3.1 STRING MANIPULATIONS:

String is a group of characters. String is defined as array of characters without a null char to terminate the array. Any string declared represents an object of java.lang.String class. This string is an immutable string object stored in memory. Each time altered version of an existing string, a new **String** object is created that contains the modifications keeping the original string unchanged. This approach is used because fixed, immutable strings can be implemented more efficiently than changeable ones. For those cases in which a modifiable string is desired, Java provides two options: java.lang. **String Buffer** and java.lang.**String Builder**. Both hold strings that can be modified after they are created.

3.1.2 String

Any string variable declared represents an object of Java.lang.String class. A string can be declared using a sequence of characters enclosed in double quotes or can be initialized using different constructors of String class.

Constructors:

String()

Create a string object without any content.

String(char *chars*[])

Creates an string object using array of characters.

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

Creates a string object using selected range of characters from array.

String(byte *chrs*[])

Creates a string object using array of bytes.

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

Creates a string object using selected range of byte from array.

String(String *strObj*)

Creates a string object using another string.

Methods:

String S1 = “Monu, Saru, Yes Mama, Eating Sugar, No Mama!!”

String S2=”Yellow Color Yellow Color Where are you? Here I am ..”

Modifier and Type	Method and Description
char	charAt(int index) Returns the char value at the specified index. s1.charAt(2) will return ‘n’
int	compareTo(String anotherString) Compares two strings lexicographically. “ABCD”.compareTo(“ABXY”) will math A, B then calculate the difference between first unmatched character X – C ie 88-67 = 22 is the result.
int	compareToIgnoreCase(String str) Compares two strings lexicographically, ignoring case differences. Same as above but comparison will ignore case differences.
String	concat(String str) Concatenates the specified string to the end of this string. “Abcd”.concat(“Xyz”) will result “AbcdXyz”, can also be done using “+” operator.
static String	copyValueOf(char[] data) Returns a String that represents the character sequence in the array specified.

static String	<code>copyValueOf(char[] data, int offset, int count)</code> Returns a String that represents the character sequence in the array specified.
boolean	<code>endsWith(String suffix)</code> Tests if this string ends with the specified suffix. S1.endsWith("Mama") will return true. And S2.endsWith("Mama") will return false.
boolean	<code>equals(Object anObject)</code> Compares this string to the specified object. S1.equals("mypassword") will return false.
boolean	<code>equalsIgnoreCase(String anotherString)</code> Compares this String to another String, ignoring case considerations.
byte[]	<code>getBytes()</code> Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.
byte[]	<code>getBytes(String charsetName)</code> Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.
void	<code>getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)</code> Copies characters from this string into the destination character array.
int	<code>indexOf(int ch)</code> Returns the index within this string of the first occurrence of the specified character. S1.indexOf('M'); will return 0
int	<code>indexOf(String str)</code> Returns the index within this string of the first occurrence of the specified substring. S1.indexOf("Mama"); will return 15
int	<code>indexOf(int ch, int fromIndex)</code> Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index. S1.indexOf('M',5); will return 15
boolean	<code>isEmpty()</code> Returns true if, and only if, length() is 0.
int	<code>lastIndexOf(int ch)</code> Returns the index within this string of the last occurrence of the specified character.

	S1.lastIndexOf('M') will return 40
int	lastIndexOf(String str) Returns the index within this string of the last occurrence of the specified substring. S1.lastIndexOf("Mama") will return 40
int	length() Returns the length of this string. S1.length('Mama') will return 45
String	replace(char oldChar, char newChar) Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar. "a for apple".replace('a','@') will return @ for @pple
boolean	startsWith(String prefix) Tests if this string starts with the specified prefix.
boolean	startsWith(String prefix, int toffset) Tests if the substring of this string beginning at the specified index starts with the specified prefix.
String	substring(int beginIndex) Returns a new string that is a substring of this string. S1.substring(25) will return "ting Sugar, No Mama!!"
String	substring(int beginIndex, int endIndex) Returns a new string that is a substring of this string. S1.substring(25, 33) will return "ting Sugar"
char[]	toCharArray() Converts this string to a new character array.
String	toLowerCase() Converts all of the characters in this String to lower case using the rules of the default locale. S1.to LowerCase() will return "monu, saru, yes mama, eating sugar, no mama!!"
String	toUpperCase() Converts all of the characters in this String to upper case using the rules of the default locale. S1.to UpperCase() will return "MONU, SARU, YES MAMA, EATING SUGAR, NO MAMA!!"
String	trim() Returns a copy of the string, with leading and trailing whitespace omitted. " AnuAruNash ".trim() will return "AnuAruNash"

3.1.3 String Buffer

StringBuffer class represents a mutable string that is growable and writable character sequences. StringBuffer may have characters and substrings inserted in the middle or appended to the end. StringBuffer will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth.

Constructors

StringBuffer defines these four constructors:

StringBuffer()

Creates a StringBuffer object with empty contents and has an initial capacity of 16 characters.

StringBuffer(int capacity)

Creates a StringBuffer object with empty contents with specified capacity.

StringBuffer(String str)

Creates a StringBuffer object with specified contents and has an initial capacity of 16 characters.

Modifier and Type	Method and Description
StringBuffer	append (StringBuffer sb) Appends the specified StringBuffer to this sequence.
int	capacity () Returns the current capacity.
char	charAt (int index) Returns the char value in this sequence at the specified index.
StringBuffer	delete (int start, int end) Removes the characters in a substring of this sequence.
StringBuffer	deleteCharAt (int index) Removes the char at the specified position in this sequence.
void	ensureCapacity (int minimumCapacity) Ensures that the capacity is at least equal to the specified minimum.
int	indexOf (String str) Returns the index within this string of the first occurrence of the specified substring.

int	indexOf(String str, int fromIndex) Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
StringBuffer	insert(int offset, char c) Inserts the string representation of the char argument into this sequence.
StringBuffer	insert(int offset, String str) Inserts the string into this character sequence.
int	lastIndexOf(String str) Returns the index within this string of the rightmost occurrence of the specified substring.
int	lastIndexOf(String str, int fromIndex) Returns the index within this string of the last occurrence of the specified substring.
int	length() Returns the length (character count).
StringBuffer	replace(int start, int end, String str) Replaces the characters in a substring of this sequence with characters in the specified String.
StringBuffer	reverse() Causes this character sequence to be replaced by the reverse of the sequence.
void	setCharAt(int index, char ch) The character at the specified index is set to ch.
void	setLength(int newLength) Sets the length of the character sequence.
String	substring(int start) Returns a new String that contains a subsequence of characters currently contained in this character sequence.
String	substring(int start, int end) Returns a new String that contains a subsequence of characters currently contained in this sequence.
String	toString() Returns a string representing the data in this sequence.
void	trimToSize() Attempts to reduce storage used for the character sequence.

3.1.4 String Tokenizer

The **String Tokenizer** class from `java.util` package provides the first step in parsing process, called the lexer, lexical analyzer or scanner. Parsing is the division of text into a set of discrete parts or tokens. String tokenization is a process where a string is broken into several parts. Each part is called a *token*. For example, if "Anu, Aru, Nash, Yes Mama" is a string, the discrete parts—such as, "Anu," "Aru," "Nash" "Yes" and "Mama" are tokens.

Constructors:

`StringTokenizer(String str)`

Creates an object using String to be tokenized and space as delimiter excluding space from token.

`StringTokenizer(String str, String delimiters)`

Creates an object using String to be tokenized and second argument as delimiter excluding space from token.

`StringTokenizer(String str, String delimiters, boolean delimAsToken)`

Creates an object using String to be tokenized and second argument as delimiter excluding space from token as specified by true/false.

String S1 = "Anu, Aru, Nash, Yes Mama, Eating Sugar, No Mama!!"

Stk = StringTokenizer(S1)

Method	Description
int	countTokens()
	Calculates the number of times that this tokenizer's <code>nextToken</code> method can be called before it generates an exception. <code>stk.countTokens()</code> will return 8
boolean	hasMoreElements()
	Returns the same value as the <code>hasMoreTokens</code> method.
boolean	hasMoreTokens()
	Tests if there are more tokens available from this tokenizer's string.
Object	nextElement()
	Returns the same value as the <code>nextToken</code> method, except that its declared return value is an <code>Object</code> rather than a <code>String</code> .

String	nextToken()
	Returns the next token from this string's tokenizer. stk.nextToken() will return "Anu," stk.nextToken() will return "Aru, "
String	nextToken(String delim)
	Returns the next token in this string's tokenizer's string. stk.nextToken("u,") will return "An" stk.nextToken("Mama") will return "Aru, Nash, Yes,"

3.2 PACKAGES:

3.2.1 Introduction to predefined packages (java.lang, java.util, java.io, java.sql, java.swing)

java.lang: Every java program implicitly uses a package java.lang. It contains classes and interfaces that are fundamental to all of Java programming. It is Java's most widely used package as it provides classes that are fundamental to the design of the Java programming language.

Class	Description
Boolean	The Boolean class wraps a value of the primitive type boolean in an object.
Byte	The Byte class wraps a value of primitive type byte in an object.
Character	The Character class wraps a value of the primitive type char in an object.
Class<T>	Instances of the class Class represent classes and interfaces in a running Java application.
ClassLoader	A class loader is an object that is responsible for loading classes.
Compiler	The Compiler class is provided to support Java-to-native-code compilers and related services.
Double	The Double class wraps a value of the primitive type double in an object.
Enum<E extends Enum<E>>	This is the common base class of all Java language enumeration types.

Float	The Float class wraps a value of primitive type float in an object.
Integer	The Integer class wraps a value of the primitive type int in an object.
Long	The Long class wraps a value of the primitive type long in an object.
Math	The class Math contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.
Number	The abstract class Number is the superclass of classes BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, and Short.
Object	Class Object is the root of the class hierarchy.
Package	Package objects contain version information about the implementation and specification of a Java package.
Runtime	Every Java application has a single instance of class Runtime that allows the application to interface with the environment in which the application is running.
SecurityManager	The security manager is a class that allows applications to implement a security policy.
Short	The Short class wraps a value of primitive type short in an object.
String	The String class represents character strings.
StringBuffer	A thread-safe, mutable sequence of characters.
StringBuilder	A mutable sequence of characters.
System	The System class contains several useful class fields and methods.
Thread	A <i>thread</i> is a thread of execution in a program.

ThreadGroup	A thread group represents a set of threads.
ThreadLocal<T>	This class provides thread-local variables.
Throwable	The Throwable class is the superclass of all errors and exceptions in the Java language.
Void	The Void class is an uninstantiable placeholder class to hold a reference to the Class object representing the Java keyword void.
ArithmeticException	Thrown when an exceptional arithmetic condition has occurred.
ArrayIndexOutOfBoundsException	Thrown to indicate that an array has been accessed with an illegal index.
ClassNotFoundException	Thrown when an application tries to load in a class through its string name using: The forName method in class Class.
Exception	The class Exception and its subclasses are a form of Throwable that indicates conditions that a reasonable application might want to catch.
NullPointerException	Thrown when an application attempts to use null in a case where an object is required.
NumberFormatException	Thrown to indicate that the application has attempted to convert a string to one of the numeric types, but that the string does not have the appropriate format.

java.util

Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).

Class	Description
ArrayList<E>	Resizable-array implementation of the List interface.
Arrays	This class contains various methods for manipulating arrays (such as sorting and searching).
Calendar	The Calendar class is an abstract class that provides methods for converting between a specific instant in time and a set of calendar fields such as YEAR, MONTH, DAY_OF_MONTH, HOUR, and so on, and for manipulating the calendar fields, such as getting the date of the next week.
Collections	This class consists exclusively of static methods that operate on or return collections.
Currency	Represents a currency.
Date	The class Date represents a specific instant in time, with millisecond precision.
EventObject	The root class from which all event state objects shall be derived.
Formatter	An interpreter for printf-style format strings.
GregorianCalendar	GregorianCalendar is a concrete subclass of Calendar and provides the standard calendar system used by most of the world.
LinkedList<E>	Doubly-linked list implementation of the List and Deque interfaces.
Locale	A Locale object represents a specific geographical, political, or cultural region.
Objects	This class consists of static utility methods for operating on objects.
Properties	The Properties class represents a persistent set of properties.
Random	An instance of this class is used to generate a stream of pseudorandom numbers.
ResourceBundle	Resource bundles contain locale-specific objects.
Scanner	A simple text scanner which can parse primitive types and strings using regular expressions.

Stack<E>	The Stack class represents a last-in-first-out (LIFO) stack of objects.
StringTokenizer	The string tokenizer class allows an application to break a string into tokens.
Timer	A facility for threads to schedule tasks for future execution in a background thread.
TimeZone	TimeZone represents a time zone offset, and also figures out daylight savings.
UUID	A class that represents an immutable universally unique identifier (UUID).
Vector<E>	The Vector class implements a growable array of objects.

java.io

Provides for system input and output through data streams, serialization and the file system.

Class	Description
BufferedInputStream	A BufferedInputStream adds functionality to another input stream-namely, the ability to buffer the input and to support the mark and reset methods.
BufferedOutputStream	The class implements a buffered output stream.
BufferedReader	Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.
BufferedWriter	Writes text to a character-output stream, buffering characters so as to provide for the efficient writing of single characters, arrays, and strings.
DataInputStream	A data input stream lets an application read primitive Java data types from an underlying input stream in a machine-independent way.
DataOutputStream	A data output stream lets an application write primitive Java data types to an output stream in a portable way.
File	An abstract representation of file and directory pathnames.

FileDescriptor	Instances of the file descriptor class serve as an opaque handle to the underlying machine-specific structure representing an open file, an open socket, or another source or sink of bytes.
FileInputStream	A FileInputStream obtains input bytes from a file in a file system.
FileOutputStream	A file output stream is an output stream for writing data to a File or to a FileDescriptor.
FilePermission	This class represents access to a file or directory.
FileReader	Convenience class for reading character files.
FileWriter	Convenience class for writing character files.
InputStream	This abstract class is the superclass of all classes representing an input stream of bytes.
InputStreamReader	An InputStreamReader is a bridge from byte streams to character streams: It reads bytes and decodes them into characters using a specified charset.
ObjectInputStream	An ObjectInputStream deserializes primitive data and objects previously written using an ObjectOutputStream.
OutputStream	This abstract class is the superclass of all classes representing an output stream of bytes.
OutputStreamWriter	An OutputStreamWriter is a bridge from character streams to byte streams: Characters written to it are encoded into bytes using a specified charset.
PrintWriter	Prints formatted representations of objects to a text-output stream.
Writer	Abstract class for writing to character streams.
IOException	Signals that an I/O exception of some sort has occurred.
FileNotFoundException	Signals that an attempt to open the file denoted by a specified pathname has failed.

java.sql

Provides the API for accessing and processing data stored in a relational database using the Java programming language.

Class	Description
Driver interface	Every JDBC driver must implement the Driver interface.
DriverManager	Driver Manager is the backbone of the JDBC architecture The DriverManager class is responsible for loading JDBC drivers and creating Connection objects.
Connection interface	A connection (session) with a specific database. SQL statements are executed and results are returned within the context of a connection.
Statement	The Statement interface executes SQL statements.
PreparedStatement	The PreparedStatement interface allows programs to precompile SQL statements for increased performance.
ResultSet	The ResultSet interface represents a database result set, allowing programs to access the data in the result set.
ResultSetMetaData	An object that can be used to get information about the types and properties of the columns in a ResultSet object. This interface provides meta information about the data underlying a particular ResultSet.
SQLException	A SQLException object is thrown by any JDBC method that encounters an error.
CallableStatement	The interface used to execute SQL stored procedures.

javax.swing

The javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.

Class	Description
JComponent	Super class of all component classes in swing.
JFrame	Creates a top class container to hold GUI components.
JApplet	Create a browser based GUI application called applet.
JLabel	The class is used to create a label.
JButton	The JButton class is used to create a push button.
TextField	The class is used to create a text input output component
JRadioButton	The class is used to create a option button component.
JList	The class is used to create a list control.

3.2.2 User Defined Packages

Packages are the used to segregate classes into meaningful groups. Java puts a class file in package at run time and locates the class from there. Java uses file system directories to store packages. The **package** statement defines a name space in which classes are stored. If you omit the **package** statement, the class names are put into the default package, which has no name.

This is the general form of the **package** statement:

Package *pkg*;

eg: package mypack;

class A {}

class B {};

The compiler will store A.class and B.class file in mypack directory/package/namespace. Hierarchy of packages can be created using the period “.” separator.

Eg. packagemypack.source.test;

class A {}

class B {};

The compiler will store A.class and B.class file in test sub-package of source package and the source package is located in mypack directory/package.

The java run time system by default uses the current working directory as its starting point. After the current working directory the runtime system searches the –CLASSPATH environmental variable and uses the directory to locate class files. Then the run time system searches the –CLASSPATH location used with javac or java command.

3.2.3 Access specifiers

Encapsulation is the mechanism of binding together code and the data it manipulates, and keeps both safe from outside interference and misuse. Java achieves this using class and four different access levels. Public, private, no-specifier (default) and protected. A **public** Class, method and field can be accessed from any other class in the Java program, whether they are in the same package or in another package. **Private** Fields and methods can be accessed within the same class to which they belong. Using private specifier we can also achieve encapsulation which is used for hiding data. **Protected** fields and methods can only be accessed by subclasses in another package or any class within the package of protected members class. **Default i. e. if not** declared any specifier, it will follow the default accessibility level and can access class, method, or field which belongs to the same package, but not from outside this package.

Visibility	Public	Protected	Default	Private
From the same class	Yes	Yes	Yes	Yes
From any class in the same package	Yes	Yes	Yes	No
From a subclass in the same package	Yes	Yes (<i>Package, Inheritance</i>)	Yes (<i>Package</i>)	No
From a subclass outside the same package	Yes	Yes (<i>Inheritance</i>)	No	No
From any non-subclass class outside the package	Yes	No	No	No

3.3 CHAPTER SUMMARY:

The chapter help to learn the classes used in string manipulation. The different methods associated with string manipulation. String is a commonly used in many desktop and web application and has a wide range of applications. The chapter further introduced the concept of Package and access specifiers.

3.4 TEXT BOOK(S):

- 1) Herbert Schildt, Java The Complete Reference, Ninth Edition, McGraw-Hill Education, 2014

3.5 ADDITIONAL REFERENCE(S):

- 1) E. Balagurusamy, Programming with Java, Tata McGraw-Hill Education India, 2014

- 2) Programming in JAVA, 2nd Ed, Sachin Malhotra & Saurabh Choudhary, Oxford Press
- 3) Core Java – SYBSC CS – Sheth Publication – Prof. Ahtesham Shaikh, Prof. Beena Kapadia
- 3) The Java Tutorials: <http://docs.oracle.com/javase/tutorial/>

3.6 QUESTIONS:

- 1) What is a string? Write a program to accept a string as a command line argument and print its reverse.
- 2) What is a string? Explain, with example, the following methods of String class:
(i) indexOf() (ii) substring()
- 3) What is a package? Explain
- 4) List any five predefined packages in java.
- 5) What is the purpose of java.util package. List any five classes or interfaces.
- 6) What is the purpose of java.io package. List any five classes or interfaces.
- 7) What is the purpose of java.sql package. List any five classes or interfaces.
- 8) How to define user defined package? How JVM locates user defined packages? Explain.
- 9) Explain the visibility of class and their members for different access specifier.



EXCEPTION HANDLING

Unit Structure

- 4.1 Introduction
- 4.2 Types of errors
- 4.3 Exceptions
- 4.4 Syntax of Exception Handling Code
- 4.5 Multiple catch Statements
- 4.6 Using finally Statement
- 4.7 Throw and throws keyword
- 4.9 Using Exception for debugging
- 4.9 Summary
- 4.10 Textbook
- 4.11 Additional References
- 4.12 Questions

4.1 INTRODUCTION

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

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

4.2 TYPES OF ERRORS

Errors may be classified into two categories:

- Compile-time errors
- Run-time errors

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

Program 4.1 Illustration of compile-time errors

```
/*This program contains an error*/

class Error1

{

    public static void main (String[] args)

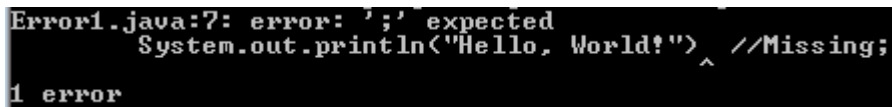
    {

        System.out.println("Hello, World!")//Missing;

    }

}
```

The Java compiler does a nice job of telling us where the errors have occurred in the program. For example, if we have missed the semicolon at the end of print statement in Program 4.1, the following message will appear on the screen.



```
Error1.java:7: error: ';' expected
    System.out.println(<'Hello, World?'> ^ //Missing;
1 error
```

We can now go to the appropriate line, correct an error and recompile the program. Sometimes, a single error may be the source of multiple errors later in the compilation. For example, use of an undeclared variable in several places will cause a series of errors of type “**undefined variable**”. In such case, we should consider the earliest errors as the major source of problem. Once we fix an error, we should recompile the program and look for other errors.

Most of the compile-time errors are due to typing mistakes. Typographical errors are hard to find, and we may have to check code word by word. The most common problems are:

- Missing semicolons
- Missing (or mismatch of) brackets in classes and methods
- Misspelling of keywords and identifiers
- Missing double quotes in strings
- Using undeclared variables
- Use of = in place of == operator and so on.

Other errors may occur because of directory paths. An error such as

javac: command not found

It means that we have not set the path correctly. We must include the path directory where the Java executables are stored.

Run-Time Errors

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

- Dividing an integer by zero
- Accessing an element that is out of the bounds of an array
- Trying to store a value into an array of an incompatible class
- Passing a parameter that is not in a valid range or value for a method
- Trying to illegally change the state of a thread
- Attempting to use a negative size for an array
- Using a null object reference as a legitimate object reference to access a method or a variable
- Converting invalid string to a number
- And many more

When such errors are encountered, Java typically generates an error message and aborts the program. Program 4.2 illustrates how a run-time error causes termination of execution of the program.

Program 4.2 Illustration of run-time errors

```
class Error2
{
    public static void main(String[] args)
    {
        int x = 10;
        int y = 5/0;
        int z = 5/0;
        int    a = x/(y-z);           //Division by zero
        System.out.println("a=" +x);
        int b = x/(y+z);
        System.out.println("b=" +y);

    }
}
```

Program 4.2 is syntactically correct and therefore does not cause any problem during compilation. However, during execution, it displays the following message and stops without executing remaining statements.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
at Error2.main(Error2.java:11)
```

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

4.3 EXCEPTIONS

An exception is a condition caused by a run-time error in the program. When the Java interpreter encounters an error such as dividing an integer by zero, it creates and throws an exception object (i.e., informs us that an error has occurred). If the exception object is not caught and handled properly, the interpreter will display an error message as shown in the output of Program 4.2 and will terminate the program.

If we want our program to continue with the execution of the remaining code, then we should try to catch the exception object thrown by the error condition and then display an appropriate message for taking corrective actions. This task is known as **exception handling**.

The purpose of exception handling is to detect and report an “exceptional circumstance” so that appropriate action can be taken. Error handling code performs the following tasks:

1. Find the problem (**Hit** the exception).
2. Inform that an error has occurred (**Throw** the exception)
3. Receive the error information (**Catch** the exception)
4. Take corrective actions (**Handle** the exception)

Error handling code consists of two segments, one to detect errors and to throw exceptions and the other to catch exceptions and take appropriate actions.

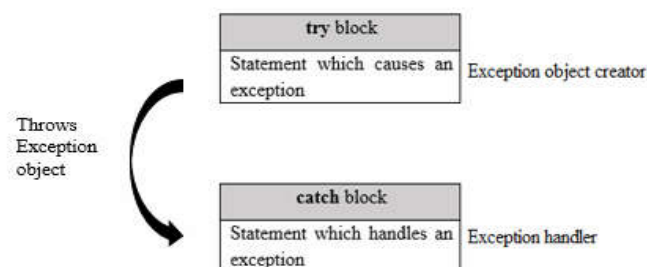
While writing programs, we must check for places in the program where an exception could be generated. Some common exceptions are listed in Table 4.1

Table 4.1 Common java Exceptions

Exception Type	Cause of Exception
ArithmeticException	It is caused by math errors such as division by zero
ArrayIndexOutOfBoundsException	Caused by bad array indexes
ArrayStoreException	Caused when a program tries to store the wrong type of data in an array
FileNotFoundException	Caused by an attempt to access a nonexistent file
IOException	Caused by general I/O failures, such as inability to read from a file
NullPointerException	Caused by referencing a null object
NumberFormatException	Caused when a conversion between strings and number fails
OutOfMemoryException	Caused when there's not enough memory to allocate a new object
SecurityException	Caused when an applet tries to perform an action not allowed by the browser's security setting
StackOverflowException	Caused when the system runs out of stack space
StringIndexOutOfBoundsException	Caused when a program attempts to access a nonexistent character position in a string.

4.4 SYNTAX OF EXCEPTION HANDLING CODE

The basic concepts of Exception handling are throwing an exception and catching it. This is illustrated in Fig. 4.1

**Fig 4.1 Exception handling mechanism**

Java uses a keyword `try` to preface a block of code that is likely to cause an error condition and “throw” an exception. The catch block is added immediately after the try block. A catch block “catches” the exception “thrown” by the try block and handles it appropriately.

```

.....

.....

try
{
    statement ;           //generates an exception
}

catch (Exception- type e)
{
    statement ;    //processes the exception
}

.....

.....

```

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

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

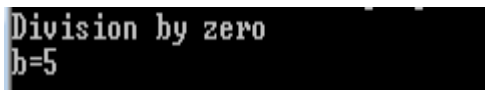
The catch statement works like a method definition. A single parameter, which is reference to the **exception object** is thrown (by the try block). If the catch parameter matches with the type of exception object, then the exception is caught and statements in the catch block will be executed. Otherwise,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

Program 4.3 illustrates the use of try and catch blocks to handle an arithmetic exception. Note that program 4.3 is a modified version of Program 4.2.

Program 4.3 Using try and catch for exception handling

```
class Error3
{
    public static void main(String[] args)
    {
        int x = 10;
        int y = 5;
        int z = 5;
        try
        {
            int a = x/(y-z);    //Exception here
        }
        catch(ArithmeticException e)
        {
            System.out.println("Division by zero");
        }
        int b = x/(y+z);
        System.out.println("b=" + y);
    }
}
```

Program 4.3 displays the following output:



```
Division by zero
b=5
```

Note that the program did not stop when an exception is caused inside try block. Exception is caught by catch block and it prints the error message, and then continues the execution, as if nothing has happened. Compare with the output of Program 4.2 which did not give the value of y.

Program 4.4 shows another example of using exception handling mechanism.

```
class TryCatchExample
{

    public static void main(String[] args)
    {
        try
        {
            int arr[]={1,3,5,7};
            System.out.println(arr[4]); //may throw exception
        }
        catch(ArrayIndexOutOfBoundsException e) // handling an array
        exception
        {
            System.out.println("Array index doesnt exist");
        }

    }

}
```

Output:

```
Array index doesnt exist
```

In this program we have array which contains 4 elements i.e arr[0], arr[1], arr[2], arr[3]. We are printing arr[4] which doesn't exist in the array list. Hence *ArrayIndexOutOfBoundsException* is caused by try block and caught by catch block.

4.5 MULTIPLE CATCH STATEMENTS

It is possible to have more than one catch statement in the catch block as illustrate below:

```
.....
.....
try
{
    statement;           //generates an exception
}
```

```

catch (Exception- Type-1 e)
{
    statement;          // processes exception type 1
}
catch (Exception- Type-2 e)
{
    statement;          // processes exception type 2
}
catch (Exception- Type-3e)
{
    statement;          // processes exception type N
}

```

.....

When an exception in a **try** block is generated, the Java treats the multiple **catch** statements like cases in a switch statement. The first statement whose parameter matches with the exception object will be executed, and the remaining statements will get skipped.

Note that Java does not require any processing of the exception at all. We can simply have a catch statement with an empty block to avoid program abortion.

Example:

```
catch (Exception e);
```

Here, the catch statement simply ends with a semicolon, which does nothing. This statement will catch an exception and then ignore it.

Program 4.5 Using multiple catch blocks

```

class MultipleCatchBlock
{
    public static void main(String[] args)
    {
        try
        {
            int a[]=new int[5];

            System.out.println(a[10]); //doesnt exist
        }
    }
}

```

```

        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("ArrayIndexOutOfBoundsException
occurs");
        }
        catch(ArithmeticException e)
        {
            System.out.println("Arithmetic Exception occurs");
        }
        catch(Exception e)
        {
            System.out.println("Parent Exception occurs");
        }
        System.out.println("rest of the code");
    }
}

```

Program 4.5 uses a chain of catch blocks and, when run, produces the following output:

```

ArrayIndexOutOfBoundsException occurs
rest of the code

```

Note that the array element `a[10]` does not exist because array `a` is defined to have only five elements, `a[0]`, `a[1]`, `a[2]`, `a[3]`, `a[4]`. Therefore, the index 10 is outside the array boundary thus causing the block

catch (ArrayIndexOutOfBoundsException e)

to catch and handle the error. Remaining catch blocks are skipped.

4.6 USING *FINALLY* STATEMENT

Java supports **finally** statement that can be used to handle an exception that is not caught by any of the previous catch statements. **finally** block can be used handle any exception generated within a **try** block. It can be added immediately after the **try** block or after the last catch block shown as follows:

```

try                                try
{                                  {
    .....                          .....
    .....                          .....
}                                  }

```



```

finally
{
    .....
    .....
}

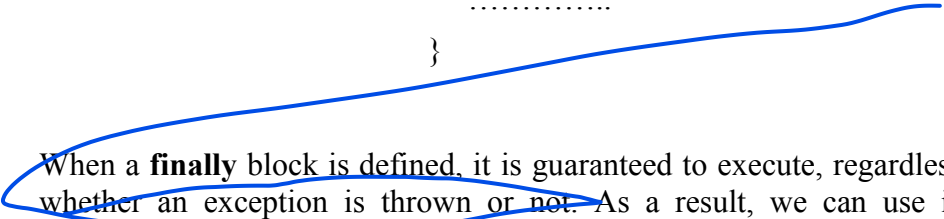
catch(.....)
{
    .....
    .....
}

catch (.....)
{
    .....
    .....
}

.
.
.

finally
{
    .....
    .....
}

```



When a **finally** block is defined, it is guaranteed to execute, regardless of whether an exception is thrown or not. As a result, we can use it to perform operations such as closing files and releasing system resources.

In Program 4.5, we may include the last statements inside a finally block as shown below:

Program 4.6 Using finally blocks

```

class FinallyBlock
{
    public static void main(String[] args)
    {
        try
        {
            int a[] = new int[5];
            System.out.println(a[10]); //doesnt exist
        }
    }
}

```

```

catch(ArrayIndexOutOfBoundsException e)
{
    System.out.println("ArrayIndexOutOfBoundsException occurs");
}
catch(ArithmeticException e)
{
    System.out.println("Arithmetic Exception occurs");
}
catch(Exception e)
{
    System.out.println("Parent Exception occurs");
}
finally
{
    System.out.println("rest of the code");
}
}

```

This will produce the same output.

```

ArrayIndexOutOfBoundsException occurs
rest of the code

```

4.7 THROWING OUR OWN EXCEPTIONS

The Java throw keyword is used to throw an exception explicitly.

We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. We can do this by using the keyword throw as follows:

throw new Throwable_subclass;

Examples:

```
throw new ArithmeticException( );
```

```
throw new NumberFormatException( );
```

In Program 4.6, we have created validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the `ArithmeticException` otherwise print a message welcome to vote.

Program 4.6 Throwing our own exception

```
class TestThrow
{
    public static void validate(int age)        //validate(age=10)
    {
        if(age<18)                //10<18
        {
            //throw Arithmetic exception if not eligible to vote
            throw new ArithmeticException("not eligible for voting");
        }
        else
        {
            System.out.println("You are eligible for voting");
        }
    }
    public static void main(String args[])
    {
        validate(10); //calling the function
        System.out.println("Rest of the code...");
    }
}
```

A run of Program 4.6 produces:

```
Exception in thread "main" java.lang.ArithmeticException: not eligible for votin
g
    at TestThrow.validate(TestThrow.java:10)
    at TestThrow.main(TestThrow.java:20)
```

Program 4.7 Java throws keyword

```
import java.io.*;

class Main1
```

```

{
// declaring the type of exception

    public static void findFile() throws IOException
    {
// code that may generate IOException

        File newFile = new File("test.txt");

        FileInputStream stream = new FileInputStream(newFile);
    }

    public static void main(String[] args)
    {
        try
        {
            findFile();
        }
        catch (IOException e)
        {
            System.out.println(e);
        }
    } }

```

Output:

```

java.io.FileNotFoundException: test.txt (The system cannot find the file speci
ed)

```

Program 4.8 Throwing our own exception


```

import java.lang.Exception;

class MyException extends Exception
{
    MyException(String message)
    {
        super(message);
    }
}

```

```
class OwnException
{
    public static void main (String args[])
    {
        int x = 5, y = 1000;
        try
        {
            float z = (float) x / (float) y;
            if(z < 0.01)
            {
                throw new MyException("Number is too small");
            }
        }
        catch(MyException e)
        {
            System.out.println("Caught my exception");
            System.out.println(e.getMessage());
        }
        finally
        {
            System.out.println("I am always here");
        }
    }
}
```

Output:A screenshot of a terminal window showing the output of the Java program. The text is displayed on three lines: "Caught my exception", "Number is too small", and "I am always here".

```
Caught my exception
Number is too small
I am always here
```

The object `e` which contains the error message “Number is too small” is caught by the catch block which then displays the message using the `getMessage()` method.

Note that Program 4.8 also illustrates the use of finally block. The last line of output is produced by the finally block.

4.8 USING EXCEPTION FOR DEBUGGING

As we have seen, the exception-handling mechanism can be used to hide errors from rest of the program. It is possible that the programmers may misuse this technique for hiding errors rather than debugging the code. Exception handling mechanism can be effectively used to locate the type and place of errors. Once we identify an error, we must try to find out why these errors occurred before we coverup them with exception handlers.

4.9 SUMMARY

A good program does not produce unexpected results. We should incorporate features that could check for potential problem spots in programs and guard against program failures. Exceptions in Java must be handled carefully to avoid any program failures.

In this chapter we have discussed the following:

- ✓ What exceptions are
- ✓ try, catch and finally block
- ✓ How to catch and handle different types of exceptions.
- ✓ How to throw system exceptions

4.10 TEXTBOOK(S):

- 1) Herbert Schildt, Java The Complete Reference, Ninth Edition, McGraw-Hill Education, 2014

4.11 ADDITIONAL REFERENCE(S):

- 1) E. Balagurusamy, Programming with Java, Tata McGraw-Hill Education India, 2014
- 2) Programming in JAVA, 2nd Ed, Sachin Malhotra & Saurabh Choudhary, Oxford Press

4.12 QUESTIONS:

- 1) What is Exception? Explain the types of exceptions.
- 2) What is error? Explain the types of error.
- 3) How the exceptions are handled in Java?



MULTITHREADING

Unit Structure

- 5.1 Introduction
- 5.2 Creating threads
- 5.3 Extending the thread Class
- 5.4 Stopping and Blocking a thread
- 5.6 Life Cycle of A thread
- 5.6 Using thread Methods
- 5.7 Synchronization in Java
- 5.8 Summary
- 5.9 Textbook
- 5.10 Additional Reference(s)
- 5.11 Questions

5.1 INTRODUCTION

Those who are familiar with the modern operating systems (Windows 10) may recognize that they can execute several programs simultaneously. This ability is known as multitasking. In system's terminology, it is called multithreading.

Multithreading is a conceptual programming paradigm where a program (process) is divided into two or more subprograms (processes), which can be implemented in parallel. This is similar to dividing one task into subtasks and assigning them to different people for execution independently and simultaneously. For example, one subprogram can display an animation on the screen while another may build the next animation to be displayed.

In most computers, there is only a single processor and therefore, in reality, the processor does only one thing at a time. However, the processor switches between the processes so fast that it appears to human beings that all of them are being executed simultaneously. Java programs that we have seen and discussed so far contain only a single sequential flow of control. This is what happens when we execute a normal program. The program begins, runs through a sequence of executions, and finally ends. At any given point of time, there is only one statement under execution.

A thread is similar to a program that has a single flow of control. It has a beginning, a body, and an end, and executes commands sequentially. All main programs in our earlier (previous chapters) examples can be called single-threaded programs. Every program will have at least one thread as shown in Fig. 5.1

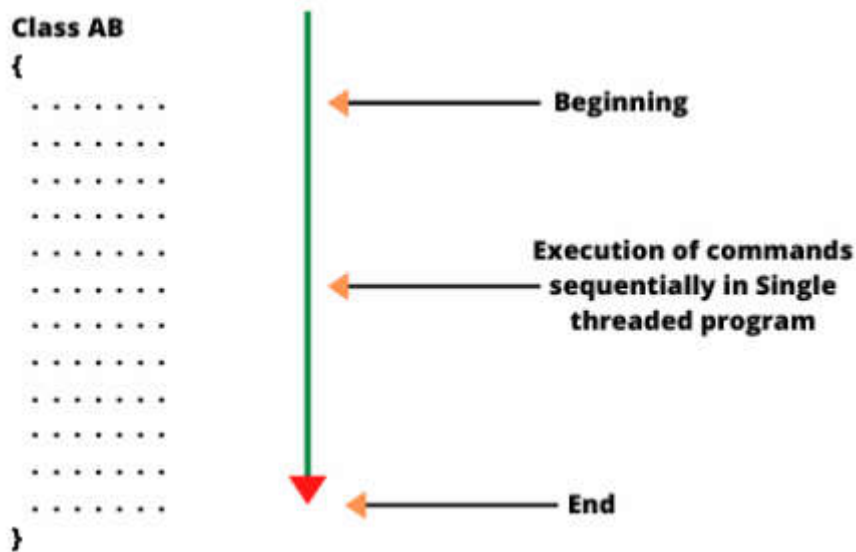


Fig. 5.1 Single-threaded program

A unique property of Java is its support for multithreading. That is, Java enables us to use multiple flows of control in developing programs. Each flow of control may be thought of as a separate tiny program (or module) known as a thread that runs in parallel with each other as shown in Figure 5.2.

A program that contains multiple flows of control is known as multithreaded program. Fig. 5.2 illustrates a Java program with four threads, one main and two others. The main method module is main thread, which is designed to create and start the other two threads, namely Thread A, Thread B.

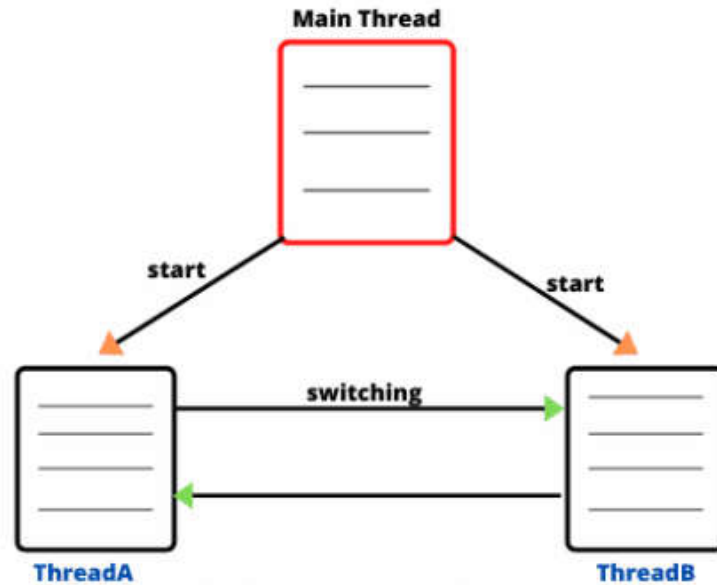


Fig. 5.2 A Multithreaded program

Once initiated by the main thread, the threads A, B run concurrently and share the resources jointly. It is like people living in joint families and sharing certain resources among all of them. Since threads in Java are subprograms of a main application program and share the same memory space, they are known as **lightweight threads** or **lightweight processes**.

It is important to remember that '**threads running in parallel**' does not really mean that they are running at the same time. Since all the threads are running on a single processor, the flow of execution is shared between the threads. The Java interpreter handles the switching of control between the threads in such a way that it appears they are running concurrently.

Multithreading is a powerful programming tool that makes Java distinctly different from its fellow programming languages. Multithreading enables programmers to do multiple things at same time. They can divide a long program (containing operations that are conceptually concurrent) into threads and execute them in parallel. For example, we can send print command into the background and continue to perform some other task in the foreground. This approach would considerably improve the speed of our programs.

Any application we are working on that requires two or more things to be done at the same time is probably a best one for use of threads.

5.2 CREATING THREADS

Creating threads in Java is simple. Threads are implemented in the form of objects that contain a method called **run()**. The **run()** method is the heart and soul of any thread. It makes up the entire body of a thread and is the

only method in which the thread's behaviour can be implemented. A typical **run()** method would appear as follows:

```
public void run()
{
.....
.....
(statement for implementing thread)
.....
.....
}
```

The **run()** method should be invoked by an object of the concerned thread. This can be achieved by creating the thread and initiating it with the help of another thread method called **start()**.

A new thread can be created in two ways.

1. **By creating a thread class:**

Define a class that extends Thread class and override its **run()** method with the code required by the thread.

2. **By converting a class to a thread:**

Define a class that implements Runnable interface. The Runnable interface has only one method, **run()**, that is to be defined in the method with the code to be executed by the thread.

The approach to be used depends on class which we have created, and what it requires. If it needs to extend another class, then we have no choice but to implement the Runnable interface, since Java classes cannot have two super classes.

5.3 EXTENDING THE THREAD CLASS

We can make our class runnable as a thread by extending the **class java.lang.Thread**. This gives us access to all the thread methods directly. It includes the following steps:

1. Declare the class as extending the **Thread** class.
2. Implement the **run()** method that is responsible for executing the sequence of code that the thread will execute.
3. Create a thread object and call the **start()** method to initiate the thread execution.

❖ **Declaring the Class**

The **Thread** class can be extended as follows:

```
class TestThread extends Thread
{
    .....
    .....
    .....
}
```

Now have a new type of thread **TestThread**.

❖ **Implementing the run() Method**

The **run()** method has been inherited by the class **TestThread**. We must override this method in order to implement the code to be executed by our thread. The basic implementation of **run()** is as follows:

```
public void run( )
{
    .....
    ..... //Thread code here
    .....
    .....
}
```

When we start any new thread, Java calls the thread's **run()** method, so it is the **run ()** where all the action takes place.

❖ **Starting New Thread**

To create and run an instance of our thread class, we will write:

```
TestThread t1 = new TestThread();

t1.start();    // invokes run( ) method
```

The first line instantiates a new object of class **TestThread**. Note that this statement just creates the object. The thread that will run this object is not yet running. The thread is in a **newborn** state.

The second line calls the **start ()** method causing the thread to move into the runnable state. Then, the Java runtime will schedule the thread to run by invoking its **run ()** method. Now, the thread is in the **running** state.

❖ An Example of Using the Thread Class

Program 5.1 illustrates the use of Thread class for creating and running threads in an application. In program we have created two threads A and B for undertaking two different tasks. The **main** method in the **ThreadTest1** class also constitutes another thread which we may call the "main thread".

The main thread dies at the end of its **main** method. However, before it dies. it creates and starts other two threads A, B.

We can start a thread as follows:

```
A t1 = new A();
```

```
t1.start();
```

Immediately after the thread A is started, there will be two threads running in the program: the main thread and the thread A.

The **start()** method returns back to the main thread immediately after invoking the **run()** method, thus the allowing the main thread to start the thread B.

Program 5.1 Creating threads using the thread class

class A extends **Thread**

```
{  
    public void run()  
    {  
        for (int i =1; i<=5; i++)  
        {  
            System.out.println("Thread A: i=" +i);  
        }  
        System.out.println("Exit from A");  
    }  
}
```

class B extends **Thread**

```
{  
    public void run()  
    {  
        for (int j =1; j<=5; j++)  
        {  
            System.out.println("Thread B: j=" +j);  
        }  
    }  
}
```

```

        System.out.println("Exit from B");
    }
}

class Threadtest1
{
    public static void main(String args[])
    {
        A t1 = new A();
        B t2 = new B();

        t1.start();           //start first thread
        t2.start();           //start second thread
    }
}

```

Output:

First run

```

Thread A: i=1
Thread B: j=1
Thread A: i=2
Thread B: j=2
Thread A: i=3
Thread B: j=3
Thread A: i=4
Thread A: i=5
Exit from A
Thread B: j=4
Thread B: j=5
Exit from B

```

Second run

```

Thread A: i=1
Thread A: i=2
Thread A: i=3
Thread A: i=4
Thread B: j=1
Thread A: i=5
Exit from A
Thread B: j=2
Thread B: j=3
Thread B: j=4
Thread B: j=5
Exit from B

```

```
Thread A: i=1
Thread A: i=2
Thread A: i=3
Thread A: i=4
Thread A: i=5
Thread B: j=1
Thread B: j=2
Thread B: j=3
Thread B: j=4
Thread B: j=5
Exit from B
Exit from A
```

By the time the main thread has reached the end of its main method, there are a total of **three** separate threads running in parallel.

We have simply initiated **two** new threads and started them. We did not hold on to them further. They are running concurrently on their own. Note that the outputs from the threads are not sequential. They do not follow any specific order.

They are running independently of one another and each executes whenever it has a chance. Remember, once the threads started. We cannot decide with certainty the order in which they may execute statements. Note a second run and third run has a different output sequence.

5.4 STOPPING AND BLOCKING A THREAD

Stopping a Thread

Whenever we want to stop a thread from running further, we may do **so** by calling **stop()** method, like:

```
aThread.stop();
```

This statement causes the thread to move to the dead state. A thread will also move to the dead state automatically when it reaches the end of its method. The **stop()** method may be used when the premature death of a thread is desired.

Blocking a Thread

A thread can also be **suspended temporarily** or blocked from entering into the runnable and subsequently running state by using either of the following thread methods:

```
sleep()    // blocked for a specified time
```

```
suspend()  // blocked until further orders
```

```
wait()     // blocked until certain condition occurs
```

These methods cause the thread to go into the **blocked** (or **not-runnable**) state.

The thread will return to the runnable state when the specified time is elapsed in the case of **sleep()**.

The **resume()** method is invoked in the case of **suspend()**, and the **notify()** method is called in the case of **wait()**.

5.5 LIFE CYCLE OF A THREAD

During the lifetime of a thread, it can enter many states. It includes:

1. Newborn state
2. Runnable state
3. Running state
4. Blocked state
5. Dead state

A thread is always in one of these five states. It can move from one state to another via a variety of ways as shown in Fig. 5.3.

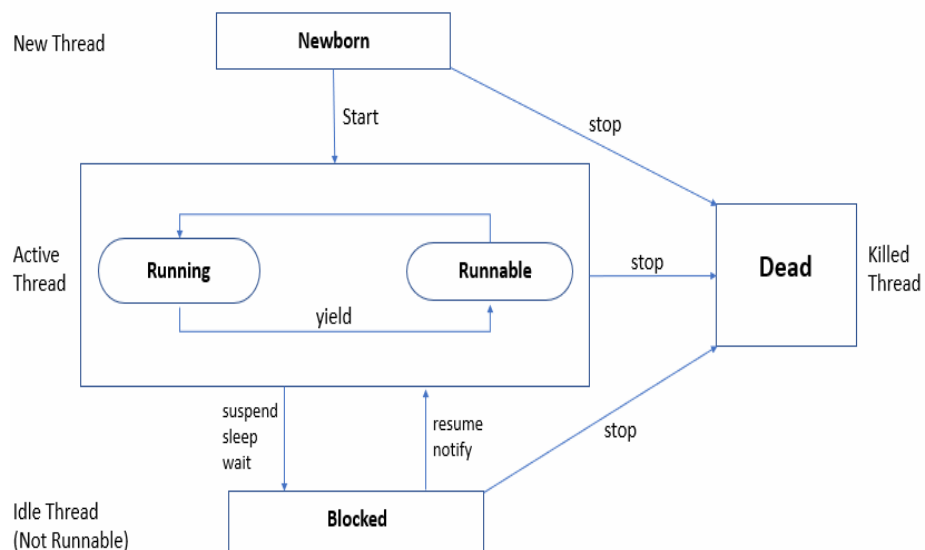


Fig. 5.3 State transition diagram of a thread

Newborn State

When we create a thread object, the thread is born and is said to be in newborn state. The thread is not yet scheduled for running. At this state, we can do only one of the following things with it:

- ✓ We can schedule it for running using **start()** method.
- ✓ We can kill it using **stop()** method.

If scheduled, it moves to the runnable state (Fig. 5.4). If we attempt to use any other method at this stage, an exception will be thrown.

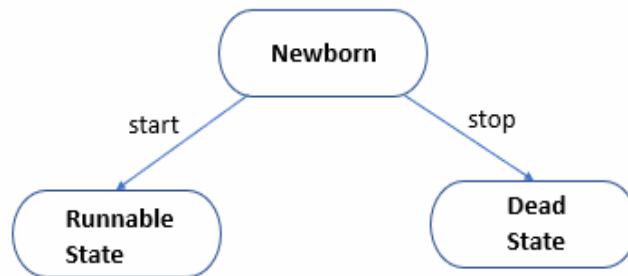


Fig. 5.4 scheduling a newborn thread

Runnable State

The runnable state means that the thread is ready for execution and is waiting for the availability of the processor, i.e the thread has joined the queue of threads that are waiting for execution. If all threads have equal priority, then they are given time slots for execution in first- come, first-serve manner. The thread that relinquishes control joins the queue at the end and again waits for its turn.

However, if we want a thread to relinquish control to another thread of equal priority before its turn comes, we can do so by using the **yield()** method. (Fig. 5.5)

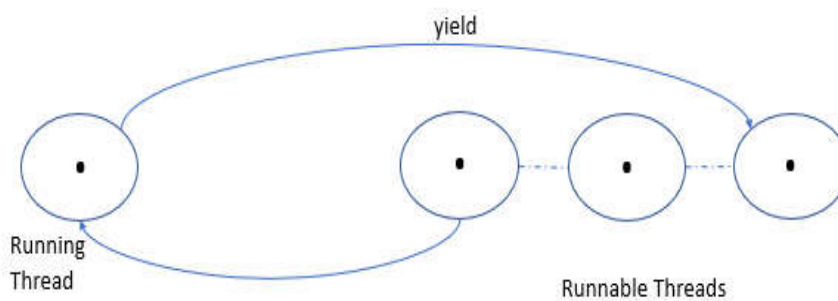


Fig. 5.5 Relinquishing control using yield() method

Running State

Running means that the processor has given its time to the thread for its execution. The thread runs until it relinquishes control on its own or it is pre-empted by a higher priority thread.

A running thread may relinquish its control in one of the following situations.

- 1) It has been suspended by using **suspend()** method. A suspended thread can be revived by using the **resume()** method.

This approach is useful when we do not want to kill a thread but want to suspend it for some time due to certain reason.

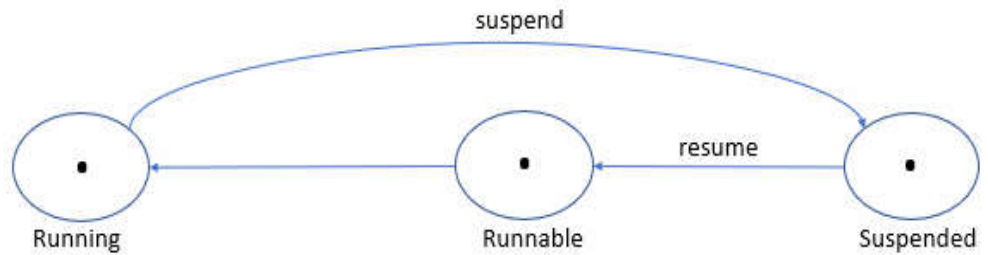


Fig. 5.6 Relinquishing control using suspend() method

2) It has been made to sleep. We can make a thread to sleep for a specified time period using the method **sleep (time)** where time is in milliseconds.

It means that the thread is out of the queue during this time period. As soon as this time period is elapsed, the thread re-enters the runnable state.

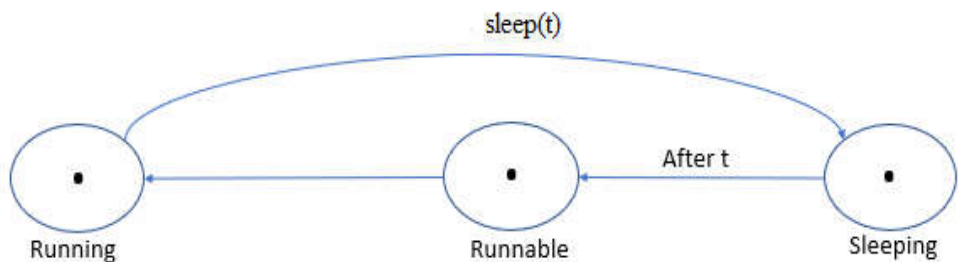


Fig. 5.7 Relinquishing control using sleep() method

3) It has been told to wait until some event occurs. It is done using the **wait()** method.

The thread can be scheduled to run again using the **notify()** method.

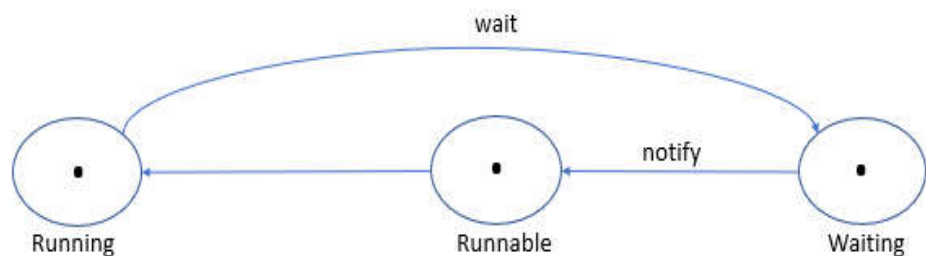


Fig. 5.8 Relinquishing control using wait() method

A thread is said to be in blocked state when it is prevented from entering the runnable state and subsequently the running state.

It happens when the thread is suspended, sleeping or waiting in order to satisfy certain requirements.

A blocked thread is considered “not runnable” but it is not dead and therefore fully qualified to run again.

Dead State

Every thread has a life cycle. A running thread ends its life when it completes executing its **run ()** method. It is a natural death.

However, we can kill it by sending the stop message to it at any state thus causing a premature death. A thread can be killed as soon as it is born, or while it is running, or even when it is in “not runnable” (blocked) condition.

5.6 USING THREAD METHODS

Thread class methods can be used to control the behaviour of a thread. We have already used the methods **start()** and **run()** in program 5.1. There are methods that can move a thread from one state to another.

Program 5.2 illustrates the use of **yield()**, **sleep()**, and **stop()** methods.

Program 5.2 Use of yield(), stop(), and sleep() methods

class A extends Thread

```
{  
    public void run()  
    {  
        for (int i =1; i<=5; i++)  
        {  
            if(i==1) yield();  
            System.out.println("Thread A: i=" +i);  
        }  
        System.out.println("Exit from A");  
    }  
}
```

class B extends Thread

```
{  
  
    public void run()  
    {  
        for (int k =1; k<=5; k++)  
        {  
            System.out.println("Thread B: k=" +k);  
            if(k==1)  
            {  
                try  
                {  
                    sleep(2000);  
                }  
                catch(Exception e)  
                {  
                }  
            }  
        }  
  
        System.out.println("Exit from B");  
    }  
}  
  
class ThreadMethods  
{  
    public static void main(String args[])  
    {  
        A t1 = new A();  
        B t2 = new B();  
        System.out.println("Start thread A");  
        t1.start();  
        System.out.println("Start thread B");  
        t2.start();  
    }  
}
```

```
System.out.println("End of main thread");
```

Multithreading

```
}  
  
}
```

Output:

```
Start thread A  
Start thread B  
End of main thread  
Thread A: i=1  
Thread B: k=1  
Thread A: i=2  
Thread A: i=3  
Thread A: i=4  
Thread A: i=5  
Exit from A  
Thread B: k=2  
Thread B: k=3  
Thread B: k=4  
Thread B: k=5  
Exit from B
```

Program 5.2 uses the **yield()** method in thread A at the iteration `i=1`. Therefore, the thread A, although started first, has relinquished its control to the thread B.

The thread B started sleeping after executing for loop only once.

When it woke up (after 2000 milliseconds), the other thread has already completed its runs and therefore was running alone.

The main thread died much earlier than the other two threads.

5.7 SYNCHRONIZATION IN JAVA

So far, we have seen threads that use their own data and methods provided inside their **run()** methods. What happens when they try to use data and methods outside themselves? In such situations, they may compete for the same resources and may lead to serious problems.

For example, one thread may try to **read** a record from a file while another is still writing to the same file. Depending on the situation, we may get strange results.

Java provides a way to overcome this problem using a technique known as **synchronization**.

In case of Java, the keyword **synchronized** helps to solve such problems by keeping a watch on such locations. For example, the method that will read information from a file and the method that will update the same file may be declared as **synchronized**.

Example:

```

synchronized void update()
{
    .....
    .....                //code here is synchronized
    .....
}

```

When we declare a method synchronized, Java creates a “monitor” and hands it over to the thread that calls the method first time. As long as the thread is holding the monitor, no other thread can enter the synchronized section of the code. A monitor is like a key and the thread that holds the key can only open the lock.

It is also possible to mark a block of code as **synchronized** as shown below:

```

synchronized (lock-object)
{
    .....                //code here is synchronized
    .....
}

```

Whenever a thread completes its work of using synchronized method (or block of code), it will hand over the monitor to the next thread that is ready to use the same resource.

A deadlock situation may occur when two or more threads are waiting to gain control of a resource. Due to some reason, the condition on which the waiting threads rely on to gain control does not happen.

For example, assume that the thread X must access Method1 before it can release Method2, but the thread Y cannot release Method1 until it gets hold of Method2. Because these are **mutually exclusive** conditions, a deadlock occurs. The code below illustrates this:

Thread X

```

synchronized method2 ()
{
    synchronized method1()
    {
        .....
    }
}

```

```

        .....
    }
}
Thread Y
synchronized method1 ()
{
    synchronized method2 ()
    {
        .....
        .....
    }
}

```

5.8 SUMMARY

A thread is a single line of execution within a program. Multiple threads can run concurrently in any single program.

A thread is created either by sub classing the Thread class or implementing the **Runnable** interface. Careful application of multithreading will considerably improve the execution speed of Java programs.

5.9 TEXTBOOK

Herbert Schildt, Java The Complete Reference, Ninth Edition, McGraw-Hill Education, 2014

5.10 ADDITIONAL REFERENCE(S)

1. E. Balagurusamy, Programming with Java, Tata McGraw-Hill Education India, 2014
2. Programming in JAVA, 2nd Ed, Sachin Malhotra & Saurabh Choudhary, Oxford Press

5.11 QUESTIONS

1. What is thread? Explain the life cycle of thread.
2. Explain the synchronization of thread.
3. Write a java program to implement the concept of thread.



I/O STREAMS

Unit Structure

6.1 Introduction

6.2 Types of Streams

6.2.1 Byte Stream

6.2.2 Character Stream

6.3 Java Input Stream Class

6.3.1 Java FileInputStream Class

6.3.2 Java ByteArray Input Stream Class

6.4 Java Output Stream Class

6.4.1 Java File Output Stream Class

6.4.2 Java Byte Array Output Stream Class

6.5 Java Reader and Writer

6.6 Summary

6.7 Textbooks

6.8 Questions

6.1 INTRODUCTION

Java I/O (Input and Output) is used *to process the input and produce the output*.

Java uses the concept of a stream to make I/O operation fast. The **java.io** package contains all the classes required for input and output operations.

CONCEPT OF STREAMS

A stream is a sequence of data. In Java, a stream is composed of bytes.

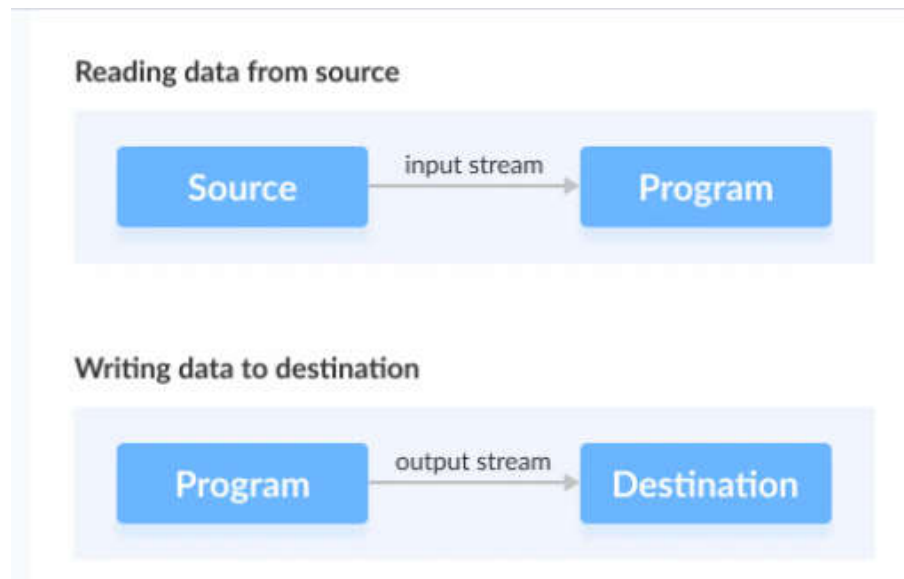
It's called a stream because it is like a stream of water that continues to flow.

In Java, streams are the sequence of data that are read from the source and written to the destination. An **input stream** is used to read data from the source and, an **output stream** is used to write data to the destination.

```
class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello, World!");
    }
}
```

For example, in above HelloWorld program, we have used **System.out** to print a string. Here, the **System.out** is a type of **output stream**.

Similarly, there are input streams to take input.



6.2 TYPES OF STREAMS

Depending upon the data a stream holds, it can be classified into following types:

- ✓ Byte Stream
- ✓ Character Stream

6.2.1 BYTE STREAM

Byte stream is used to read and write a single byte (8 bits) of data.

All byte stream classes are derived from base abstract classes called **InputStream** and **OutputStream**.

6.2.2 CHARACTER STREAM

Character stream is used to read and write a single character of data.

All the character stream classes are derived from base abstract classes **Reader** and **Writer**.

6.3 JAVA INPUTSTREAM CLASS

The **InputStream** class of the **java.io** package is an abstract superclass that represents an input stream of bytes.

Since **InputStream** is an abstract class, it is not useful by itself. However, its subclasses can be used to read data.

❖ SUBCLASSES OF INPUTSTREAM

In order to use the functionality of **InputStream**, we can use its following subclasses.

- ✓ **FileInputStream**
- ✓ **ByteArrayInputStream**
- ✓ **ObjectInputStream**
- ✓ **BufferedInputStream**

❖ CREATE AN INPUTSTREAM

In order to create an **InputStream**, we must import the **java.io.InputStream** package first.

Once we import the package, here is how we can create the input stream.

// Creates an InputStream

InputStream object1 = new FileInputStream();

Here, we have created an input stream using **FileInputStream**. It is because **InputStream** is an abstract class. Hence, we cannot create an object of **InputStream**.

❖ METHODS OF INPUTSTREAM

The **InputStream** class provides different methods that are implemented by its subclasses.

Some of the commonly used methods are:

- ✓ **read()** –
It reads one byte of data from the input stream.
- ✓ **read(byte[] array)** –
It reads bytes from the stream and stores in the specified array.
- ✓ **available()** –
It returns the number of bytes available in the input stream.
- ✓ **mark()** –

It marks the position in the input stream up to which data has been read.

✓ **reset()** –

It returns the control to the point in the stream where the mark was set.

✓ **markSupported()** –

It checks if the mark() and reset() method is supported in the stream.

✓ **skip()** –

It skips and discards the specified number of bytes from the input stream.

✓ **close()** –

It closes the input stream.

6.3.1 JAVA FileInputStream CLASS

The **FileInputStream** class of the **java.io** package can be used to read data (in bytes) from files. It extends the **InputStream** abstract class.

❖ CREATE A FileInputStream

In order to create a **FileInputStream**, we must import the **java.io.FileInputStream** package first. Once we import the package, here is how we can create a file input stream in Java.

- Using the path to file

```
FileInputStream input = new FileInputStream(stringPath);
```

Here, we have created an input stream that will be linked to the file specified by the path.

- Using an object of the file

```
FileInputStream input = new FileInputStream(File fileObject);
```

Here, we have created an input stream that will be linked to the file specified by fileObject.

❖ METHODS OF FileInputStream

The **FileInputStream** class provides implementations for different methods present in the **InputStream** class.

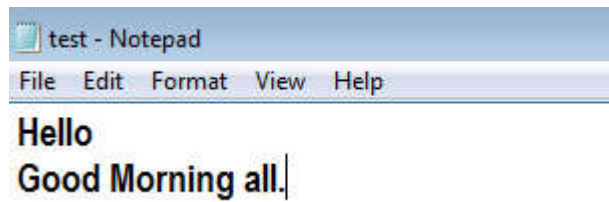
read() Method

- ✓ **read()** - It reads a single byte from the file.
- ✓ **read(byte[] array)** –It reads the bytes from the file and stores in the specified array.
- ✓ **read(byte[] array, int start, int length)** –It reads the number of bytes equal to length from the file and stores in the specified array starting from the position start.

EXAMPLE: INPUTSTREAM USING FileInputStream

Here is how we can implement **Input Stream** using the **File Input Stream** class.

Suppose we have a file named **test.txt** with the following content.



Let's try to read this file using **FileInputStream** (a subclass of **InputStream**).

Program 6.2

```
import java.io.FileInputStream;
import java.io.InputStream;

class Program
{
    public static void main(String args[])
    {
        byte[] array = new byte[100];

        try
        {
            InputStream input = new FileInputStream("test.txt");
            System.out.println("Available bytes in the file: " + input.available());

            // Read byte from the input stream
            input.read(array);

            System.out.println("Data read from the file: ");

            // Convert byte array into string
            String data = new String(array);

            System.out.println(data);

            // Close the input stream
            input.close();
        }
        catch (Exception e)
```

```

    {
e.printStackTrace();
    }
}
}
}

```

Output:

```

E:\programs>javac Program.java
E:\programs>java Program
Available bytes in the file: 24
Data read from the file is:
Hello
Good Morning all.

```

In the above example, we have created an input stream using the **FileInputStream** class.

The input stream is linked with the file test.txt.

```
InputStream input = new FileInputStream("test.txt");
```

To read data from the test.txt file, we have implemented these two methods.

```
input.read(array);    // to read data from the input stream
```

```
input.close();        // to close the input stream
```

We have used the **available()** method to check the number of available bytes in the file input stream.

6.3.2 JAVA ByteArrayInputStream CLASS

The **ByteArrayInputStream** class of the **java.io** package can be used to read an array of input data (in bytes). It extends the **InputStream** abstract class.

❖ CREATE A BYTEARRAYINPUTSTREAM

In order to create a byte array input stream, we must import the **java.io.ByteArrayInputStream** package first.

Once we import the package, we can create an input stream as follows:

```
// Creates a ByteArrayInputStream that reads entire array
```

```
ByteArrayInputStream input = new ByteArrayInputStream(byte[]  
arr);
```

Here, we have created an input stream that reads entire data from the arr array.

However, we can also create the input stream that reads only some data from the array.

// creates a ByteArrayInputStream that reads a portion of array

```
ByteArrayInputStream input = new ByteArrayInputStream(byte[]  
arr, int start, int length);
```

Here the input stream reads the number of bytes equal to **length** from the array starting from the **start** position.

❖ METHODS OF BYTEARRAYINPUTSTREAM

The **ByteArrayInputStream** class provides implementations for different methods present in the **InputStream** class.

read() Method

✓ **read()** –

It reads the single byte from the array present in the input stream.

✓ **read(byte[] array)** –

It reads bytes from the input stream and stores in the specified array.

✓ **read(byte[] array, int start, int length)** –

It reads the number of bytes equal to length from the stream and stores in the specified array starting from the position start.

EXAMPLE: BYTEARRAYINPUTSTREAM TO READ DATA

Program 6.3

```
import java.io.ByteArrayInputStream;

class Program3
{
    public static void main(String[] args)
    {

        // Creates an array of byte
        byte[] array = {1, 2, 3, 4};

        try
        {
            ByteArrayInputStream input = new ByteArrayInputStream(array);

            System.out.print("The bytes read from the input stream: ");

            for(int i= 0; i<array.length; i++)
            {

                // Reads the bytes
```

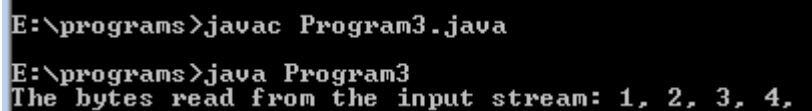
```

        int data = input.read();
        System.out.print(data + ", ");
    }
    input.close();
}

catch(Exception e)
{
    e.printStackTrace();
}
}
}

```

Output:



```

E:\programs>javac Program3.java
E:\programs>java Program3
The bytes read from the input stream: 1, 2, 3, 4,

```

In the above example, we have created a byte array input stream named input.

ByteArrayInputStream input = new ByteArrayInputStream(array);

Here, the input stream includes all the data from the specified array.

To read data from the input stream, we have used the read() method.

6.4 JAVA OUTPUTSTREAM CLASS

The **OutputStream** class of the **java.io** package is an abstract superclass that represents an output stream of bytes.

Since **OutputStream** is an abstract class, it is not useful by itself. However, its subclasses can be used to write data.

❖ SUBCLASSES OF OUTPUTSTREAM

In order to use the functionality of **OutputStream**, we can use its subclasses. Some of them are:

- ✓ **FileOutputStream**
- ✓ **ByteArrayOutputStream**
- ✓ **ObjectOutputStream**
- ✓ **BufferedOutputStream**

❖ CREATE AN OUTPUTSTREAM

In order to create an **OutputStream**, we must import the **java.io.OutputStream** package first. Once we import the package, here is how we can create the output stream.

```
// Creates an OutputStream
```

```
OutputStream object = new FileOutputStream();
```

Here, we have created an object of output stream using **File Output Stream**. It is because **OutputStream** is an abstract class, so we cannot create an object of **OutputStream**.

❖ METHODS OF OUTPUTSTREAM

The **OutputStream** class provides different methods that are implemented by its subclasses. Some of the methods are as follows:

✓ **write()** -

It writes the specified byte to the output stream.

✓ **write(byte[] array)** -

It writes the bytes from the specified array to the output stream.

✓ **flush()** -

It forces to write all data present in output stream to the destination.

✓ **close()** -

It closes the output stream.

6.4.1 JAVA FileOutputStream CLASS

The **FileOutputStream** class of the **java.io** package can be used to write data (in bytes) to the files. It extends the **OutputStream** abstract class.

❖ CREATE A FILEOUTPUTSTREAM

In order to create a file output stream, we must import the **java.io.FileOutputStream** package first.

Once we import the package, we can create a file output stream in Java as follows.

1. Using the path to file

```
// Including the boolean parameter
```

```
FileOutputStream output = new FileOutputStream(String path,  
boolean value);
```

```
// Not including the boolean parameter
```

```
FileOutputStream output = new FileOutputStream(String path);
```

Here, we have created an output stream that will be linked to the file specified by the path.

Also, value is an optional boolean parameter. If it is set to true, the new data will be appended to the end of the existing data in the file. Otherwise, the new data overwrites the existing data in the file.

2. Using an object of the file

FileOutputStream output = new FileOutputStream(File fileObject);

Here, we have created an output stream that will be linked to the file specified by fileObject.

EXAMPLE: OUTPUTSTREAM USING FILEOUTPUTSTREAM

Program 6.4

```
import java.io.FileOutputStream;
import java.io.OutputStream;
class Program1
{
    public static void main(String args[])
    {
        String data = "This is a line of text inside the file.";
        try
        {
            OutputStream out = new FileOutputStream("output.txt");
            // Converts the string into bytes
            byte[] dataBytes = data.getBytes();
            // Writes data to the output stream
            out.write(dataBytes);
            System.out.println("Data is written to the file successfully.");
            // Closes the output stream
            out.close();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```



```

    }
}
}

```

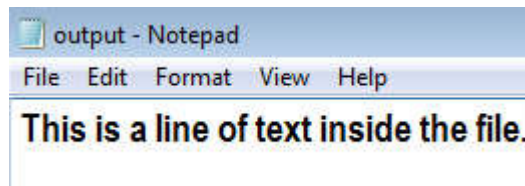
When we run the program, the **output.txt** file is filled with the following content.

```

E:\programs>javac Program1.java
E:\programs>java Program1
Data is written to the file successfully.

```

Output.txt file filled with content



In the above example, we have created an output stream using the **FileOutputStream** class. The output stream is now linked with the file **output.txt**.

```
OutputStream out = new FileOutputStream("output.txt");
```

To write data to the **output.txt** file, we have implemented following methods.

```
out.write();    // To write data to the file
```

```
out.close();   // To close the output stream
```

6.4.2 Java ByteArrayOutputStream CLASS

The **ByteArrayOutputStream** class of the **java.io** package can be used to write an array of output data (in bytes). It extends the **OutputStream** abstract class.

❖ CREATE a **ByteArrayOutputStream**

In order to create a byte array output stream, we must import the **java.io.ByteArrayOutputStream** package first.

Once we import the package, here is how we can create an output stream.

```
// Creates a ByteArrayOutputStream with default size
```

```
ByteArrayOutputStream out = new ByteArrayOutputStream();
```

Here, we have created an output stream that will write data to an array of bytes with default size 32 bytes. However, we can change the default size of the array.

```
// Creating a ByteArrayOutputStream with specified size
```

```
ByteArrayOutputStream out = new ByteArrayOutputStream(int  
size);
```

Here, the size specifies the length of the array.

❖ **METHODS OF ByteArrayOutputStream**

The **ByteArrayOutputStream** class provides the implementation of the different methods present in the **OutputStream** class.

write() Method

✓ **write(int byte)** –

It writes the specified byte to the output stream.

✓ **write(byte[] array)** –

It writes the bytes from the specified array to the output stream.

✓ **write(byte[] arr, int start, int length)** –

It writes the number of bytes equal to length to the output stream from an array starting from the position start.

✓ **writeTo(ByteArrayOutputStream out1)** –

It writes the entire data of the current output stream to the specified output stream.

EXAMPLE: ByteArrayOutputStream TO WRITE DATA

Program 6.5

```
import java.io.ByteArrayOutputStream;

class Program4
{
    public static void main(String[] args)
    {
        String data = "Hello all";

        try
        {
            // Creates an output stream

            ByteArrayOutputStream out = new ByteArrayOutputStream();

            byte[] array = data.getBytes();

            // Writes data to the output stream

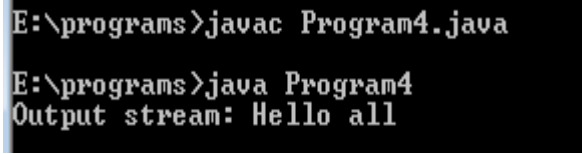
            out.write(array);
```

```

        // Retrieves data from the output stream in string format
        String streamData = out.toString();
        System.out.println("Output stream: " + streamData);

        out.close();
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
}
}

```

Output:


```

E:\programs>javac Program4.java
E:\programs>java Program4
Output stream: Hello all

```

In the above example, we have created a byte array output stream named output.

ByteArrayOutputStream output = new ByteArrayOutputStream();

To write the data to the output stream, we have used the **write()** method.

To close the output stream, we can use the **close()** method.

6.5 JAVA READER

Java Reader is an abstract class for reading character streams. The only methods that a subclass must implement are **read(char[], int, int)** and **close()**.

Some of the implementation class are **BufferedReader**, **CharArrayReader**, **FilterReader**, **InputStreamReader**,

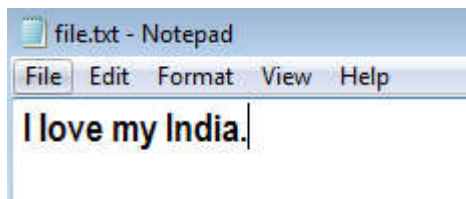
PipedReader, **StringReader**

Program 6.6

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

```
{  
    public static void main(String[] args)  
    {  
        try  
        {  
            Reader reader = new FileReader("file.txt");  
            int data = reader.read();  
            while (data != -1)  
            {  
                System.out.print((char) data);  
                data = reader.read();  
            }  
            reader.close();  
        } catch (Exception ex)  
        {  
            System.out.println(ex.getMessage());  
        }  
    }  
}
```

It is **file.txt** having content "I love my India."



Output:

```
E:\programs>javac ReaderExample.java  
E:\programs>java ReaderExample  
I love my India.
```

Java Writer

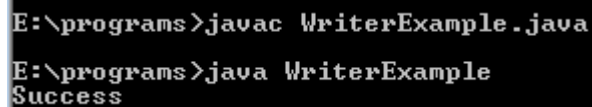
It is an abstract class for writing to character streams. The methods that a subclass must implement are `write(char[], int, int)`, `flush()`, and `close()`.

Program 6.7

```
import java.io.*;

public class WriterExample
{
    public static void main(String[] args)
    {
        try
        {
            Writer w = new FileWriter("output1.txt");
            String content = "I love my country";
            w.write(content);
            w.close();
            System.out.println("Success");
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

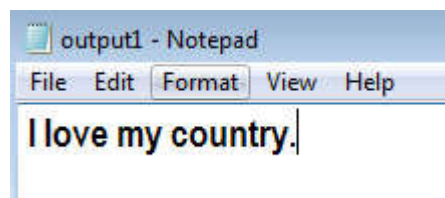
Output:



```
E:\programs>javac WriterExample.java
E:\programs>java WriterExample
Success
```

Content is written to file output1.txt

output1.txt:



6.6 SUMMARY

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 memory arrays.

Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects. Some streams simply pass on data; others manipulate and transform the data in useful ways.

6.7 TEXTBOOKS

Herbert Schildt, Java The Complete Reference, Ninth Edition, McGraw-Hill Education, 2014

6.8 QUESTIONS

- 1) What is stream? Explain the types of stream.
- 2) Explain the difference between input & output stream class?



NETWORKING

Unit Structure

- 7.1 Introduction
- 7.2 Java networking terminology
- 7.3 Java networking classes
- 7.4 Java networking interfaces
- 7.5 Java socket programming
 - 7.5.1 Socket class
 - 7.5.2 ServerSocket class
- 7.6 Summary
- 7.7 Reference
- 7.8 Questions

7.1 INTRODUCTION

Networking is a concept of connecting two or more computing devices together so that we can share resources. Java socket programming provides facility to share data between different computing devices.

The **java.net package** supports two protocols:

1. TCP:

TCP stands for **Transmission Control Protocol**. It provides reliable communication between the sender and receiver.

It is used along with the Internet Protocol referred as **TCP/IP**. TCP is a connection-oriented protocol which means that once a connection is established, data can be transmitted in two directions. This protocol is typically used over the Internet Protocol. Therefore, TCP is also referred to as TCP/IP.

TCP has built-in methods to examine for errors and ensure the delivery of data in the order it was sent, making it a complete protocol for transporting information like still images, data files, and web pages.

2. UDP:

UDP stands for **User Datagram Protocol**. It provides a connection-less protocol service by allowing packet of data to be transferred along two or

more nodes. It allows data packets to be transmitted between different applications.

UDP is a simple Internet protocol in which error-checking and recovery services are not required. In UDP, there is no overhead for opening a connection, maintaining a connection, or terminating a connection. In UDP, the data is continuously sent to the recipient, whether they receive it or not.

7.2 JAVA NETWORKING TERMINOLOGY

Java Networking Terminologies are given as follows:

1. IP Address

An IP address is a unique address assigned to a device that distinguishes a device on the internet or a local network.

IP stands for “Internet Protocol.” It comprises a set of rules governing the format of data sent via the internet or local network. It is composed of octets. The range of each octet varies from 0 to 255.

- Range of the IP Address – 0.0.0.0 to 255.255.255.255
- IP address Example – 192.168.0.1

2. Protocol

A network protocol is an organized set of commands that define how data is transmitted between different devices in the same network. Network protocols are the reason through which a user can easily communicate with people all over the world and thus play a critical role in modern digital communications.

For Example – Transmission control protocol(TCP), File Transfer Protocol (FTP), Post Office Protocol(POP), etc.

3. MAC Address

MAC address stands for **Media Access Control address**. It is a identifier that is allocated to a NIC (Network Interface Controller/ Card). It contains a 48 bit or 64-bit address, which is combined with the network adapter. MAC address can be in hexadecimal composition. In simple words, a MAC address is a unique number that is used to track a device in a network.

4. Socket

A socket is an endpoint of a two-way communication connection between the two applications running on the network. The socket mechanism presents a method of inter-process communication (IPC) by setting named contact points between which the communication occurs. A socket is bound to a specific port number so that the TCP layer can identify the application to which the data is intended to be sent to.

5. Connection-oriented and Connection-less protocol

In a connection-oriented service, the user must establish a connection before starting the communication. When the connection is established, the user can send the message or the information, and after this, they can release the connection.

In connectionless protocol, the data is transported in one route from source to destination without verifying that the destination is still there or not or if it is ready to receive the message. Authentication is not needed in the connectionless protocol.

- **Example of Connection-oriented Protocol**

Transmission Control Protocol (TCP)

- **Example of Connectionless Protocol**

User Datagram Protocol (UDP)

6. Port Number

A port number is a way to recognize a process connecting internet or other network information when it reaches a server. The port number is used to identify different applications uniquely and behaves as a communication endpoint among applications. The port number is associated with an IP address for transmission and communication among two applications. There are 65,535 port numbers, but not all are used every day.

7.3 JAVA NETWORKING CLASSES

The **java.net** package of the Java programming language includes various classes that provide an easy-to-use means to access network resources. The classes covered in the **java.net** package are given as follows –

1. CacheRequest

This class is used in java whenever there is a need to store resources in ResponseCache. The objects of this class provide an edge for the OutputStream object to store resource data into the cache.

2. CookieHandler

This class is used in Java to implement a callback mechanism to hook up an HTTP state management policy implementation inside the HTTP protocol handler. The HTTP state management mechanism specifies the mechanism of how to make HTTP requests and responses.

3. CookieManager

This class is used to provide a precise implementation of CookieHandler. This class separates the storage of cookies from the policy surrounding

accepting and rejecting cookies. A CookieManager comprises a CookieStore and a CookiePolicy.

4. DatagramPacket

This class is used for the connectionless transfer of messages from one system to another. This class provides tools to produce datagram packets for connectionless transmission by applying the datagram socket class.

5. InetAddress

This class is used to provide methods to get the IP address of any hostname. An IP address is represented by a 32-bit or 128-bit unsigned number. InetAddress can handle both IPv4 and IPv6 addresses.

6. ServerSocket

This class is used for implementing system-independent implementation of the server-side of a client/server Socket Connection. The constructor for ServerSocket class throws an exception if it can't listen on the specified port.

For example –

It will throw an exception if the port is already in use.

7. Socket

This class is used to create socket objects that help users in implementing all fundamental socket operations. The users can implement various networking actions such as sending, reading data, and closing connections.

Each Socket object is built using **java.net.Socket** class that has been connected exactly with 1 remote host; for connecting to another host, a user must create a new socket object.

8. DatagramSocket

This class is a network socket that provides a connection-less point for sending and receiving packets. Datagram Sockets is Java's mechanism for providing network communication via UDP instead of TCP. Every packet sent from a datagram socket is individually routed and delivered. It can further be practiced for transmitting and accepting broadcast information.

9. Proxy

A proxy is a kind of tool or program or system, which serves to preserve the data of its users and computers. It behaves like a wall between computers and internet users. A Proxy Object represents the Proxy settings to be applied with a connection.

10. URL

The URL class in Java is the entry point to any available sources on the internet. A Class URL describes a **Uniform Resource Locator**, which is a signal to a “resource” on the World Wide Web.

A source can be a simple file or directory, or it can indicate a more difficult object, such as a query to a database or a search engine.

7.4 JAVA NETWORKING INTERFACES

The **java.net** package of the Java programming language includes various interfaces that provide an easy-to-use means to access network resources. The interfaces included in the **java.net** package are as follows:

1. CookiePolicy

The CookiePolicy interface in the **java.net** package provides the classes for implementing various networking applications. It decides which cookies should be accepted and which should be rejected.

In CookiePolicy, there are three pre-defined policy implementations, namely ACCEPT_ALL, ACCEPT_NONE, and ACCEPT_ORIGINAL_SERVER.

2. CookieStore

A CookieStore is an interface that describes a storage space for cookies. CookieManager combines the cookies to the CookieStore for each HTTP response and recovers cookies from the CookieStore for each HTTP request.

3. FileNameMap

The FileNameMap interface is an uncomplicated interface that implements a tool to outline a file name and a MIME type string. FileNameMap charges a filename map (known as a mimetable) from a data file.

4. SocketOption

The SocketOption interface helps the users to control the behavior of sockets. Often, it is essential to develop necessary features in Sockets. SocketOptions allows the user to set various standard options.

5. SocketImplFactory

The SocketImplFactory interface defines a factory for SocketImpl instances. It is used by the socket class to create socket implementations that implement various policies.

6. ProtocolFamily

This interface represents a family of communication protocols.

The ProtocolFamily interface contains a method known as `name()`, which returns the name of the protocol family.

7.5 JAVA SOCKET PROGRAMMING

Socket programming is a way of connecting two nodes on a network to communicate with each other. One **socket** (node) listens on a specific port at an IP, while other *socket* reaches out to the other in order to form a connection.

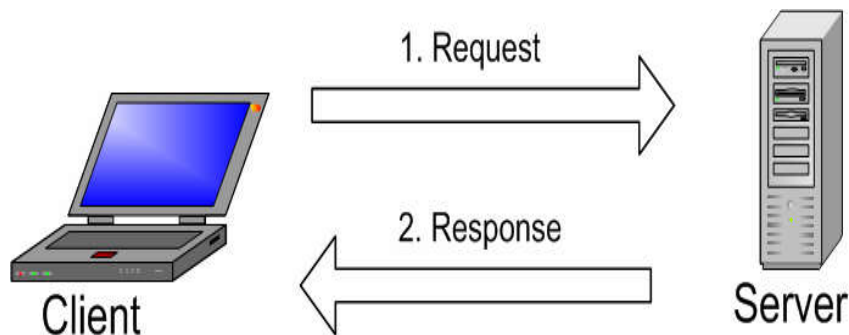


Fig. 7.1 Client- Server communication

The server forms the listener *socket* while the client reaches out to the server.

Now let's understand the core concept of Socket Programming i.e. a socket.

❖ SOCKET

A **socket** in Java is one endpoint of a two-way communication link between two programs running on the network. A **socket** is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to.



Fig. 7.2 Socket

An endpoint is a combination of an IP address and a port number. The package in the Java platform provides a class, **Socket** which implements one side of a two-way connection between your Java program and another program on the network.

The class sits on top of a platform-dependent implementation, hiding the details of any system from your Java program. By using the class instead of relying on native code, your Java programs can communicate over the network in a platform-independent fashion.

Java Socket programming can be connection-oriented or connection-less.

Socket and **ServerSocket** classes are for connection-oriented socket programming and **DatagramSocket** and **DatagramPacket** classes are used for connection-less socket programming.

The client in socket programming must know these two things:

1. IP Address of Server, and
2. Port number

A client application generates a socket on its end of the communication and strives to combine this socket with a server. When the connection is established, the server generates an object of socket class on its communication end. The client and the server can now communicate by writing to and reading from the socket.

The **java.net.Socket** class describes a socket, and the **java.net.ServerSocket** class implements a tool for the server program to host clients and build connections with them.

7.5.1 SOCKET CLASS

The **Socket class** is used to create socket objects that help the users in implementing all basic socket operations. The users can implement various networking actions such as sending data, reading data, and closing connections.

Each Socket object created using **java.net.Socket** class has been associated specifically with 1 remote host. If a user wants to connect to another host, then he must create a new socket object.

Methods of Socket Class

In Socket programming, both the client and the server have a Socket object, so all the methods under the Socket class can be invoked by both the client and the server. There are many methods in the Socket class.

Sr No.	Method	Description
1	public void connect (Socket Address host, int timeout)	It is used to connect the socket to the specified host. This method is required only when the user instantiates the Socket applying the no-argument constructor.
2	public int get Port()	It is used to return the port to which the socket is pinned on the remote machine.
3	public Inet Address get Inet Address()	It is used to return the location of the other computer to which the socket is connected.
4	public int getLocalPort()	It is used to return the port to which the socket is joined on the local machine.
5	public Socket Address get Remote Socket Address()	It returns the location of the remote socket.
6	public Input Stream get Input Stream()	It is used to return the input stream of the socket. This input stream is combined with the output stream of the remote socket.
7	public Output Stream get Output Stream()	It is used to return the output stream of the socket. The output stream is combined with the input stream of the remote socket.
8	public void close()	It is used to close the socket, which causes the object of the Socket class to no longer be able to connect again to any server.

7.5.2 SERVERSOCKET CLASS

The **ServerSocket** class is used for providing system-independent implementation of the server-side of a client/server Socket Connection.

The constructor for **ServerSocket** class throws an exception if it can't listen on the specified port. For example –

It will throw an exception if the port is already in use.

Methods of Server Socket Class:

Methods of the Server Socket class are as follows:

Sr No.	Method	Description
1	public int get Local Port()	This method of Server Socket class is used to give the port number of the server on which this socket is listening. If the socket was bound before being closed, then this method will continue to return the port number after the socket is closed.
2	public void set So Timeout (int timeout)	It is used to set the time-out value for the time in which the server socket pauses for a client during the accept () method. The timeout value should be greater than 0 otherwise, it will throw an error.
3	Public Socket accept ()	This method waits for an incoming client. This method is blocked till either a client combines to the server on the specified port or the socket times out, considering that the time-out value has been set using the setSoTimeout() method. Otherwise, this method will be blocked indefinitely.
4	public void bind (Socket Address host, int backlog)	This method is used to bind the socket to the specified server and port in the object of Socket Address. The user should use this method if he has instantiated the Server Socket using the no-argument constructor.

Program 7.1 Example of Java Socket Programming**Creating Server:**

To create the server application, we need to create the instance of **Server Socket** class.

Here, we are using 6666 port number for the communication between the client and server. You can also choose any other port number.

The accept() method waits for the client. If clients connect with the given port number, it returns an instance of Socket.

```
ServerSocket ss=new ServerSocket(6666);
```

```
Socket s= ss.accept();//establishes connection and waits for the client
```

Creating Client:

To create the client application, we need to create the instance of Socket class.

Here, we need to pass the IP address or hostname of the Server and a port number. Here, we are using "localhost" because our server is running on same system.

```
Socket s=new Socket("localhost",6666);
```

Let's see a simple example of Java socket programming where client sends a text message, server receives and prints it.

Filename: MyServer.java

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

public class MyServer
{
    public static void main(String[] args)
    {
        try
        {
            ServerSocket ss=new ServerSocket(6666);
            Socket s=ss.accept();           //establishes connection
            DataInputStream dis=new DataInputStream(s.getInputStream());
            String str=(String)dis.readUTF();
            System.out.println("message= "+str);
            ss.close();
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
}
```


Core JAVA

```
}  
}  
}
```

File: MyClient.java

```
import java.io.*;  
import java.net.*;  
public class MyClient  
{  
    public static void main(String[] args)  
    {  
        try  
        {  
            Socket s= new Socket("localhost",6666);  
            DataOutputStreamdout=new DataOutputStream(s.getOutputStream());  
            dout.writeUTF("Hello Server");  
            dout.flush();  
            dout.close();  
            s.close();  
        }  
        catch(Exception e)  
        {  
            System.out.println(e);  
        }  
    }  
}
```

To run on Terminal or Command Prompt

Open two windows one for Server and another for Client.

1. First run the Server application (MyServer.java). It will show:

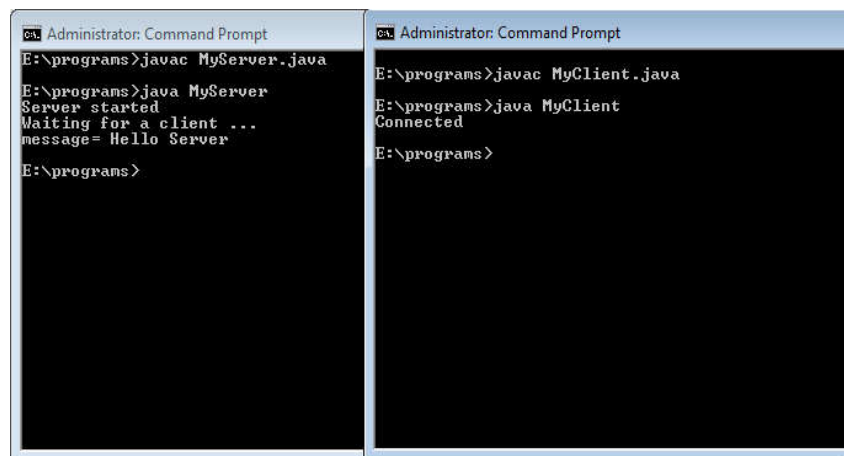
```
E:\programs>javac MyServer.java
E:\programs>java MyServer
Server started
Waiting for a client ...
```

2. Then run the Client application (MyClient.java) on another terminal. It will show:

```
E:\programs>javac MyClient.java
E:\programs>java MyClient
Connected
```

and the server accepts the client and a message will be displayed on the server console.

```
E:\programs>javac MyServer.java
E:\programs>java MyServer
Server started
Waiting for a client ...
message= Hello Server
```



Program 7.2 Example of Java Socket Programming (Read-Write both side)

In this example, client will write first to the server then server will receive and print the text.

Then server will write to the client, client will receive and print the text.

The step goes on.

File: MyServer1.java

```
import java.net.*;
```

```

import java.io.*;

class MyServer1
{
    public static void main(String args[]) throws Exception
    {
        ServerSocket ss=new ServerSocket(3333);

        Socket s=ss.accept();

        DataInputStream din=new DataInputStream(s.getInputStream());

        DataOutputStream dout=new DataOutputStream(s.getOutputStream());

        BufferedReader br=new BufferedReader(new
        InputStreamReader(System.in));

        String str="",str2="";

        while(!str.equals("stop"))
        {
            str=din.readUTF();

            System.out.println("client says: "+str);

            str2=br.readLine();

            dout.writeUTF(str2);

            dout.flush();

        }

        din.close();

        s.close();

        ss.close();

    }
}

```

File: MyClient1.java

```

import java.net.*;

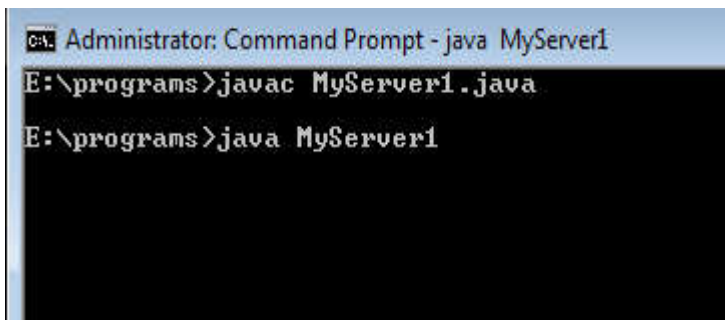
import java.io.*;

class MyClient1
{

```

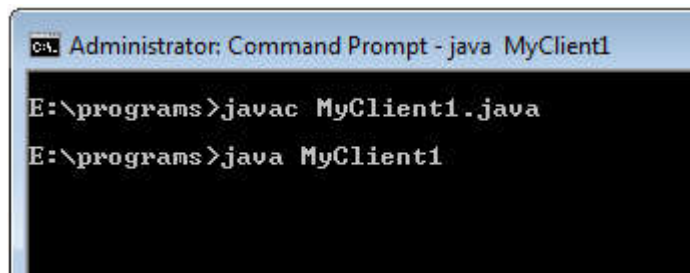
```
public static void main(String args[])throws Exception
{
    Socket s=new Socket("localhost",3333);
    DataInputStream din=new DataInputStream(s.getInputStream());
    DataOutputStreamdout=new DataOutputStream(s.getOutputStream());
    BufferedReaderbr=new BufferedReader(new
    InputStreamReader(System.in));
    String str="",str2="";
    while(!str.equals("stop"))
    {
        str=br.readLine();
        dout.writeUTF(str);
        dout.flush();
        str2=din.readUTF();
        System.out.println("Server says: "+str2);
    }
    dout.close();
    s.close();
}
```

First run the Server application. It will show:



```
ca. Administrator: Command Prompt - java MyServer1
E:\programs>javac MyServer1.java
E:\programs>java MyServer1
```

Then run the Client application on another terminal. It will show:



```

Administrator: Command Prompt - java MyClient1

E:\programs>javac MyClient1.java
E:\programs>java MyClient1

```

and the server accepts the client

Then you can start typing messages in the Server and Client window.



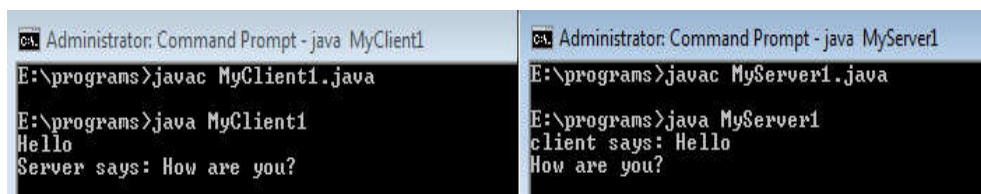
```

Administrator: Command Prompt - java MyClient1
Administrator: Command Prompt - java MyServer1

E:\programs>javac MyClient1.java
E:\programs>javac MyServer1.java

E:\programs>java MyClient1
E:\programs>java MyServer1
Hello
client says: Hello

```



```

Administrator: Command Prompt - java MyClient1
Administrator: Command Prompt - java MyServer1

E:\programs>javac MyClient1.java
E:\programs>javac MyServer1.java

E:\programs>java MyClient1
E:\programs>java MyServer1
Hello
client says: Hello
Server says: How are you?
How are you?

```

7.6 SUMMARY

Networking is a concept of connecting two or more computing devices together so that we can share resources. Java socket programming provides facility to share data between different computing devices.

7.7 REFERENCE

- 1) E. Balagurusamy, Programming with Java, Tata McGraw-Hill Education India, 2014
- 2) Programming in JAVA, 2nd Ed, Sachin Malhotra & Saurabh Choudhary, Oxford Press

7.8 QUESTIONS

- 1) Write a short note on java.net package.
- 2) What is socket? Explain the Socket Class with example.



WRAPPER CLASSES

Unit Structure

8.0 Objective

8.1 Introduction

8.2 Types of Wrapper classes

8.3 Summary

8.4 Exercise

8.5 Reference

8.0 OBJECTIVE

Objective of this chapter is to learn

1. Need of objects and primitive data types
2. How to convert primitive data types to objects and vice-versa
3. Autoboxing and unboxing feature of Java5

8.1 INTRODUCTION:

As you know Java supports primitive data types and non-primitive data types. In programming, many cases, there is need of object representation. In such standard representation primitive data types are not suitable. For example,

1. Data structures implemented by java uses collection of objects.
2. To maintain the state of the data while sending it to remote machine, java objects are serialized. Serialization is the process of converting an object type to byte stream so that data can transfer over the network. Similarly at receiver side once data is received, those byte stream data need to convert into an object. This process is called Deserialization.

A wrapper class provides the mechanism of converting primitive data type to an objects and object into primitive data types. Table 9.1 shows the primitive data types and their respective wrapper class

Table 8.1 Primitive types and their Wrappers

Primitive types	Wrapper classes
boolean	Boolean
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character

In Next session lets discuss in detail about each class.

8.2 WRAPPER CLASSES

Boolean

Methods	Description
Boolean(boolean b)	Creates the Boolean object which holds the same Boolean value as that of b Ex: boolean b=True; Boolean b1= Boolean(b)
Boolean(String b)	Creates the Boolean object which holds the same Boolean value as that of String variable b ex: String b="True"; Boolean b1= Boolean(b)
booleanbooleanValue()	Returns primitive boolean equivalent value of Boolean object boolean b=b1.booleanValue();

Program 8.1: Use of Boolean wrapper class

```

public class booeandemo{
    public static void main(String []args)
    {
        boolean b=true;
        Boolean b1=new Boolean(b);
        System.out.println("Boolean object --> "+b1);
        String s="False";
        b1=new Boolean(s);
        System.out.println("Boolean object --> "+b1);
        System.out.println("Boolean object to primitive value -->
        "+b1.booleanValue());
    }
}

```

Output:

Boolean object --> true

Boolean object --> false

Boolean object to primitive value --> false

Table 8.2 shows the listing of other wrapper classes and their methods

Table 8.2: Wrapper classes and their methods

Wrapper class	Method	Description
Integer	Integer(int intval)	Creates an Integer objects from int value
	Integer(String intval) throws NumberFormatException	Creates an Integer objects from String coded int value. Ex: String intval="10"; If intval is not in above form then it throws the NumberFormatException Refer program 8.2
	int intValue()	Return the int value present in Integer Object.

Byte	Byte(byte b) Byte(String b) throws NumberFormatException	Creates Byte objects from byte value Creates Byte objects from String coded byte value.
	Byte byteValue()	Return the byte value present in Byte Object.
Short	Short(short sh) Short(String sh) throws NumberFormatException	Creates Short objects from short value Creates Short objects from String coded short value.
	Short shortValue()	Return the short value present in Short Object.
Long	Long(long l) Long(String l) throws NumberFormatException	Creates Long objects from long value Creates Long objects from String coded long value.
	long longValue()	Return the long value present in Long Object.
Float	Float(float f) Float(String f) throws NumberFormatException	Creates Float objects from float value Creates Float objects from String coded float value.
	Float floatValue()	Return the float value present in Float Object.
Double	Double(double d) Double(String d) throws NumberFormatException	Creates Double objects from double value Creates Double objects from String coded double value.
	Double doubleValue()	Return the double value present in Double Object.
Character	Character(char ch)	Creates Character objects from char value
	Character charValue()	Return the char value present in Character Object.

Use of wrapper classes in java program will be similar as demonstrated in program 8.1.

Program 8.2: Demonstrate use of Integer Wrapper class and NumberFormatException

```
public class wrapperdemo
{
    public static void main(String []args)
    {
        String s="10";
        Integer n1=new Integer(s);
        System.out.println("Integer object --> "+n1);
        String s1="Ten";
        Integer n2=new Integer(s1);
        System.out.println("Integer value --> "+n2);
    }
}
```

Integer object --> 10

Exception in thread "main" java.lang.NumberFormatException: For input string: "Ten"

At
 java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
 at java.lang.Integer.parseInt(Integer.java:580)
 at java.lang.Integer.<init>(Integer.java:867)
 at booeandemo.main(booeandemo.java:8)

8.3 AUTOBOXING AND UNBOXING

Java 5 supports automatic conversion of primitive data type into its corresponding wrapper class. It is known as autoboxing. The reverse of it is called as unboxing. Program 8.3 shows the autoboxing and unboxing.

Program 8.3: Demo of AutoBoxing and Unboxing

```
import java.util.*;
public class wrapperdemo{
    public static void main(String []args)
    {
        int s=12;
        Integer n1=s; //autoboxing
        Integer [] n2={20,30}; //autoboxing
```

```

ArrayList<Integer> a1 = new ArrayList<Integer>();
    a1.add(n1);
    a1.add(n2[0]);
    a1.add(n2[1]);
System.out.println("Elements in arraylist are--> "+a1);
    if(n1<10) //unboxing
System.out.println(n1+" is smaller than 10 ");
    else
System.out.println(n1+" is larger than 10");
    }
}

```

Output:

Elements in arraylist are--> [12, 20, 30]

12 is larger than 10

8.4 SUMMARY:

1. For handling the collection of objects required primitive data in object form
2. Wrapper classes wrapped the primitive data and present them in object form
3. There are Long, Short, Byte, Integer, Float, Double, Character, Boolean wrapper classes in java
4. xxxValue() method used to convert xxx type object into primitive data form.

8.5 EXERCISE:

1. What is wrapper class?
2. Create a list of integer values 10,20,30,40,50.
3. What is autoboxing and unboxing?

8.6 REFERENCES:

1. H. Schildt, *Java Complete Reference*, vol. 1, no. 1. 2014.
2. E. Balagurusamy, *Programming with Java*, Tata McGraw-Hill Education India, 2014
3. Sachin Malhotra & Saurabh Choudhary, *Programming in JAVA*, 2nd Ed, Oxford Press
4. The Java Tutorials: <http://docs.oracle.com/javase/tutorial/>



COLLECTION FRAMEWORK

Unit Structure

- 9.0 Objective
- 9.1 Introduction to Collections Framework
- 9.2 Util package
- 9.3 List
- 9.4 Set
- 9.5 Map
- 9.6 List interface & its classes
 - 9.6.1 ArrayList
 - 9.6.2 LinkedList
- 9.7 Set interface & its classes
 - 9.7.1 HashSet
 - 9.7.2 TreeSet
- 9.8 Map interface & its classes
 - 9.8.1 HashMap
 - 9.8.2 TreeMap
 - 9.8.3 Iterator
- 9.9 Summary
- 9.10 Exercise
- 9.11 Reference & Bibliography

9.0 OBJECTIVE

Objective of this Chapter is to provide

- Detail insight of Collection Framework
- Use of various framework classes such as List, Set, Map in Java language
- Understanding of use of collection classes in various application

9.1 INTRODUCTION TO COLLECTIONS FRAMEWORK:

Collection means group of things. For example, collection of coins called a bunch, collection of tickets. Similarly, in computing collection of data in a one unit which helps to store, manipulate the data easily.

Collection framework is a framework which is use to represent the data and helps in manipulating the data in easy way. Each collection framework provides the methods to represent the data (Interface), manipulate the data (Implementations) and algorithms to search, sort the data efficiently.

Collection framework provides high performance by allowing the programmer to implement the defined data structures and algorithms. Low level complexities for defining the data structure and the algorithms are eliminated. Instead, use of appropriate Collection framework for defining and manipulating the group of data is required.

9.2 UTIL PACKAGE

Utility classes such as Collection framework, event, date and time, internationalization, currency, StringTokenizer are present in the java.util package. Lets see Collection framework in detail.

Collection framework provides different Interfaces for representing the data. Figure 9.1 shows the collection framework hierarchy.

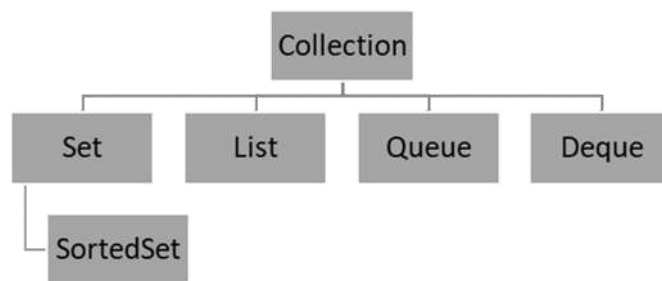


Figure 9.1: Collection Interfaces

Collection Interface: It is a super interface and provides the basic methods such as addition of element in collection, removal of element, etc. Table 9.1 shows the basic methods supported by Collection interface.

Table 9.1- Methods of Collection Interface

Method	Description
Boolean add (Object element)	Add single element into collection
Boolean remove (Object element)	Remove single element from the collection
int size()	Return the size of collection
boolean isEmpty()	Returns true if collection is empty otherwise false
boolean contains (Object element)	Returns if element is present in the collection otherwise false

Boolean contains All (Collection collection)	Returns true if collection is a subset of current collection otherwise false
boolean add All (Collection collection)	Returns true if all elements of collection are added to current collection
void remove All (Collection collection)	Returns true if all elements of collection are removed from current collection
void retain All (Collection collection)	It retains all the elements of current collection which are present in the collection and removes the elements that are not in collection (similar to intersection)
void clear()	Removes all elements from the collection
Iterator iterator()	start to finish traversal in collection is possible using Iterator interface object. This method gives reference to the Iterator Interface hence its methods can be used.

9.3 LIST INTERFACE:

List is the interface derived from Collection interface. List allows the duplicates and stores the elements in order. List maintains the position-oriented collection of objects. Table 9.2 shows the methods of list interface.

Table 9.2- Methods of List Interface

Method	Description
void add(int index, Object element)	Insert an elements at specific index location
boolean addAll(int index, Collection collection)	Insert all elements of collection at specific index location.
Object get(int index)	Returns the element present at specified index position
int indexOf(Object element)	Returns an index value of specified element
int lastIndexOf(Object element)	Returns the last index of elements specified. If the element is not present in the list then it returns -1
Object remove(int index)	Returns an element which is present at specified index position.
Object set(int index, Object element)	It replaces the element present at the specified index position with new element.
List subList(int fromIndex, int toIndex)	Returns the sub-list from- to index position

9.4 THE SET INTERFACE:

Set is the derived from Collection interface and does not allow the duplication of elements. If we try to add duplicate element using **add** method, it returns false. Set does not have any additional method. All methods are inherited from Collection interface (please refer Table 9.1).

9.5 MAP INTERFACE:

Map is not inherited from Collection interface. Map interface allows the collection of elements in the form of key-value pair. With the help of the key, value can be searched. Keys duplication is not allowed but value may be duplicated. Table 9.3 listed the methods of Map interface.

Table 9.3 shows the methods of Map interface.

Method	Description
Object put (Object key, Object value)	Insert an object in a MAP
Object remove (Object key)	Removes an object having specified key from Map
void putAll(Map mapping)	Put all the elements specified in the Map.
void clear ()	All map entries are cleared
Object get(Object key)	Returns an element whose key is mention
boolean containsKey(Object key)	Returns true if key is in Map otherwise false
boolean containsValue(Object value)	Returns true if the value is present in map otherwise false
boolean isEmpty()	Returns true if Map is empty otherwise false
int size()	Returns the size or number of elements present in the map
public Set keyset ()	Returns Set of keys present in map
public Collection values()	Returns all values in Collection form
public Set entrySet()	Returns Set of all elements in key and value pair form

**Table 9.4 shows the interfaces and their implementation classes.
Interfaces and the classes which implement them.**

Interface name	Classes
List	ArrayList
	LinkedList
Set	HashSet
	TreeSet
Map	HashMap
	TreeMap

9.6.1 ArrayList class

ArrayList class extends AbstractList class and implements List interface. ArrayList is a resizable. Arrays in java are fixed in size but arrayList allows you to create a collection of object which can be accessible like array and can grow or shrink during execution. Table 9.5 shows the constructor and methods of ArrayList class

Table 10.5 Methods of ArrayList class

Methods	Description
ArrayList()	Creates an empty ArrayList object
ArrayList(Collection c)	Creates an ArrayList object using existing collection
ArrayList(int capacity)	Creates an ArrayList object with some initial capacity
object[] toArray()	Converts ArrayList object to an array of object.

The code below demonstrates the use of ArrayList class.

Program 9.1: ArrayList class.

```
import java.util.*;

public class ArrayListDemo {
    public static void main(String args[])
    {
        // Create an array list.
        ArrayList<Integer> al = new ArrayList<Integer>();
        // Add elements to the array list.
```

```

al.add(1);
al.add(2);
al.add(3);
al.add(4);

System.out.println("Contents of ArrayList : " + al);

// Get the array.
Integer ia[] = new Integer[al.size()];
ia = al.toArray(ia);
int sum = 0;

// Sum the array.
for(int i=0;i<ia.length;i++)
    sum += ia[i];

System.out.println ("Sum of elements of an Array is: " + sum);
}
}

```

Output:

Contents of ArrayList : [1, 2, 3, 4]

Sum of elements of an Array is: 10

9.6.2 LinkedList class

The LinkedList class extends AbstractSequentialList class and implements the List, Deque, and Queue interfaces. Table 9.6 shows the constructors and the methods of LinkedList class.

Table 9.6: Methods of LinkedList class

Methods	Description
LinkedList()	Creates an empty Linkedlist object.
LinkedList (Collection c)	Creates Linked list object using existing collection elements.
void addFirst()	Add the elements in the beginning of the list
void addLast()	Add the elements at the end of the list
E peekFirst()	To obtain the first element of the list, where E is a type parameter
E peekLast()	To obtain/retrieve the last element of the list

Program to demonstrate use of LinkedList class

Program 9.2: LinkedList class

```
import java.util.*;

public class LinkedListDemo {

    public static void main(String args[]) {

        // Create a linked list.

        LinkedList<String> ll = new LinkedList<String>();

        // Add elements to the linked list.

        ll.add("Seeta");
        ll.add("Babita");
        ll.add("Deepak");
        ll.add("Keshav");
        ll.addLast("Zareena");
        ll.addFirst("Amita");
        ll.add(1, "Aarti");

        System.out.println("Original elements of list: " + ll);

        // Remove elements from the linked list.

        ll.remove("Deepak");
        ll.remove(1);

        System.out.println("list elements after deletion: " + ll);

        // Remove first and last elements.

        ll.removeFirst();
        ll.removeLast();

        System.out.println("List elements after deleting first and last: " + ll);

        // Get and set a value.

        String val = ll.get(1);
        ll.set(1, val + "_FY");
```

```

System.out.println("List after modification: " + ll);

}

}

```

Output:

Original elements of list: [Amita, Aarti, Seeta, Babita, Deepak, Keshav, Zareena]

list elements after deletion: [Amita, Seeta, Babita, Keshav, Zareena]

List elements after deleting first and last: [Seeta, Babita, Keshav]

List after modification: [Seeta, Babita_FY, Keshav]

9.7 SET INTERFACE & ITS CLASSES

9.7.1 HashSet class

HashSet extends AbstractSet and implements Set interface. HashSet uses the concept of hashing to store the elements. Key is automatically converted to hash code automatically. We could not able to access the hash code. Here HashSet does not have its own methods. They are inherited from the super class and interface it implements. HashSet does not guarantee the arrangement of the elements in sorted order. Table 9.7 lists constructors of HashSet class.

Table 9.7: Constructors of HashSet class

Methods	Description
HashSet()	HashSet is used to create a HashSet which has default initial capacity of 16 elements and fill-ratio of 0.75
HashSet(Collection c)	Create a HashSet with existing Collection object.
HashSet(int capacity)	Creates a HashSet object with initial capacity
HashSet(int capacity, float fillRatio)	Capacity is the numeric value which tells how many elements in hashSet Fillratio is the number that tells at what size, the capacity of HashSet should be increase automatically.

9.7.2 TreeSet class

Objects in TreeSet are stored in ascending order. Object retrieval time is fast. Figure 9.2 shows the class hierarchy and Table 9.8 shows the constructors of TreeSet class.

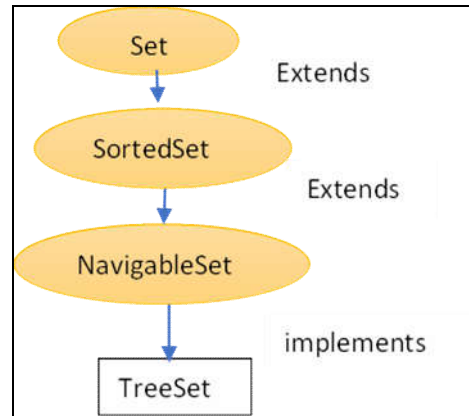


Figure 9.2: TreeSet class hierarchy

Table 9.8 Constructors of TreeSet class

Constructors	Description
TreeSet()	Creates an empty TreeSet object
TreeSet(Collection c)	Create TreeSet object from existing collection object
TreeSet(Comparator comp)	Create TreeSet object in order define by the comparator object.
TreeSet(SortedSet ss)	Create TreeSet object from existing SortedSet object

Program 9.3 demonstrates how to create an object an object and use the methods of TreeSet class and HashSet class.

Program 9.3 : Demo of TreeSet class and HashSet class

```

import java.util.*;

public class SetClassDemo {
    public static void main(String args[]) {
        // Create a hash set.
        HashSet<String> hs = new HashSet<String>();
        // Add elements to the hash set.
        hs.add("1");
        hs.add("21");
        hs.add("31");
    }
}
  
```

```

hs.add("3");
hs.add("Epsilon");
hs.add("Omega");

System.out.println("HashSet-->--> Storing is as per the hash-code
generated automatically "+hs);

TreeSet<String> ts = new TreeSet<String>();
ts.add("Beta");
ts.add("Alpha");
ts.add("12");
ts.add("zerba");
ts.add("Eta");
ts.add("56");

System.out.println(" TreeSet-->Elements are in sorted Order "+ts);
}
}

```

Output:

HashSet--> Storing is as per the hash-code generated automatically[1, 3, Epsilon, Omega, 31, 21]

TreeSet-->Elements are in sorted Order [12, 56, Alpha, Beta, Eta, zerba]

9.8 MAP INTERFACE & ITS CLASSES

Figure 9.3 shows the hierarchy of the classes implementing Map interface. Lets see in detail the implementation and use of these classes.

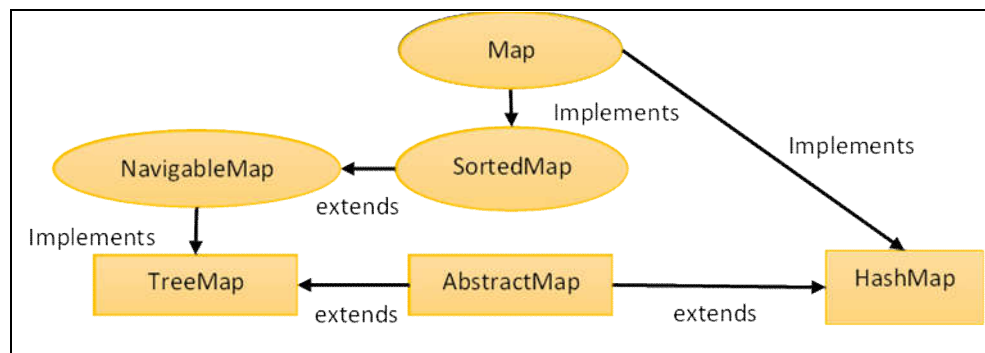


Figure 9.3: Map Interfaces and classes

9.8.1 HashMap class:

HashMap class extends AbstractMap and implements Map interface. Table 9.9 shows constructors of HashMap class.

Table 9.9 Constructors of HashMap class

Methods	Description
HashMap()	Create empty HashMap
HashMap(Map m)	Create HashMap with existing Map elements.
HashMap(int capacity)	Create a HashMap with initial capacity.
HashMap(int capacity, float fillRatio)	Create a HashMap with initial capacity and the fill ratio which is in between 0.0 to 1.0

Following code demonstrate the use of HashMap

Program 9.4: Demonstrate the use of HashMap

```
import java.util.*;

public class Student {
    public static void main(String args[]) {
        // Create a hash map.
        HashMap<String, Integer> hm = new HashMap<String, Integer>(2);
        // Put elements to the map
        hm.put("Akash", new Integer(34));
        hm.put("Mahesh", new Integer(123));
        hm.put("Prakash", new Integer(137));
        System.out.println(hm.get("Akash"));
        // Deposit 1000 into John Doe's account.
        Integer oldmark = hm.get("Akash");
        hm.put("Akash", oldmark + 10);
        System.out.println("Akash new marks : " + hm.get("Akash"));
    }
}
```

Output:

34

Akash new marks : 44

9.8.2 TreeMap class

The TreeMap class is used to implement Map interface. The class is defined by extending AbstractMap and implementing the NavigableMap interface. The Objects are stored in a tree structure. In TreeMap key/value pairs are stored in sorted order and it allows fast retrieval of elements. Table 9.10 shows the constructors and methods of TreeMap class.

Table 9.10 Constructors of TreeMap class

Methods	Description
TreeMap()	Constructor used to create an empty tree map and keys will be sorted in natural order
TreeMap(Comparator comp)	Constructor creates an empty tree-based map and keys will be sorted using the Comparator object.
TreeMap(Map m)	constructor creates a tree map using existing map elements and keys will be sorted in natural order.
TreeMap(SortedMap sm)	Constructor creates a tree map using existing SortedMap, and keys will be sorted in the same order as in SortedMap sm.

Program shows the demonstration of how to use TreeMap.

Program 9.5: Demonstration of TreeMap

```
import java.util.TreeMap;

public class TreeMapClass {

    public static void main(String[] args) {

        // TODO code application logic here

        TreeMap tm=new TreeMap();

        //adding Elements in TreeMap

        tm.put("BSCIT", new Integer(120));

        tm.put("MSCIT", new Integer(40));

        tm.put("BSCCS", new Integer(40));

        System.out.println("TreeMAp Elements");

    }

}
```

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

```
//remove element from TreeMap
```

```
tm.remove("BSCIT");
```

```
System.out.println("After Removing TreeMAp Element");
```

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

```
}
```

```
}
```

Output

TreeMAp Elements

```
{BSCCS=40, BSCIT=120, MSCIT=40}
```

After Removing TreeMAp Element

```
{BSCCS=40, MSCIT=40}
```

9.8.3 Iterator:

This Iterator Interface is used for any collection to traverse through collection. It is a cursor which is iterated through the collection to access or to remove the element from the collection. Table 9.11 shoes the methods of Iterator Interface.

Table 9.11: Methods of Iterator

Methods	Description
boolean hasNext()	Returns true if collection has more elements in iteration
Object next().	Returns the next elements in the iteration process of collection.
void remove()	Removes the element from collection the last elements accessed by iterator .

Program 9.6: Demonstration of Iterator Class

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

public class IteratorExample {
    public static void main(String[] args)
    {
        ArrayList<String> names = new ArrayList<String>();
        names.add("Akash");
        names.add("Sunil");
        names.add("Anil");
        names.add("Sania");
        names.add("Nirmala");

        // Iterator to iterate the cityNames
        Iterator iterator = names.iterator();

        System.out.println("Names elements : ");
        while (iterator.hasNext())
            System.out.print(iterator.next() + " ");
    }
}
```

Output:

Names elements:

Akash Sunil Anil Sania Nirmala

9.9 SUMMARY

- Collection interfaces are foundation interfaces for managing the group of objects.
- Java collection framework supports the List, Set, Map.
- List elements are accessed with index.
- ArrayList, are the classes of List interface.

- Set collects unique elements. TreeSet uses the concept of tree to store the data elements whereas HashSet uses hashing techniques.
- HashSet stores the elements in hashTable.
- Map interface allows the mapping between key and value. Map is implemented using HashMap and TreeMap classes.
- HashMap allows storing null elements but the keys for those elements must be different.

9.10 EXERCISE

1. What is collection in java? How to define collection of objects?
2. Write a java code to define the list of 10 students' information.
3. Write the java code to sort the above students' collection on basis of their first name.
4. How to convert ArrayList elements to Array?
5. What is the difference between HashSet and ArrayList?
6. What is a difference between ArrayList and LinedList?
7. How to access the elements from HashMap? Write the java code for the same.

9.11 REFERENCES

1. H. Schildt, Java Complete Reference, vol. 1, no. 1. 2014.
2. E. Balagurusamy, Programming with Java, Tata McGraw-Hill Education India, 2014
3. Sachin Malhotra & Saurabh Choudhary, Programming in JAVA, 2nd Ed, Oxford Press
4. The Java Tutorials: <http://docs.oracle.com/javase/tutorial/>



INNER CLASSES

Unit Structure

- 10.0 Objective
- 10.1 Introduction
- 10.2 Inner class/nested class
- 10.3 Method Local inner class
- 10.4 Static inner class
- 10.5 Anonymous inner class
- 10.6 Summary
- 10.7 Exercise
- 10.8 References

10.0 OBJECTIVE:

Objective of this chapter is

- Learn the use of inner classes
- Learn the different type of inner classes
- Learn implementation of classes with the help of anonymous class.

10.1 INTRODUCTION

Inner class is the class which is a member of other class. Inner Class could not be accessed from outside world. They are accessible to only class inclosing it. When the developer does not want the outside world to access some classes then the concept of inner class is useful.

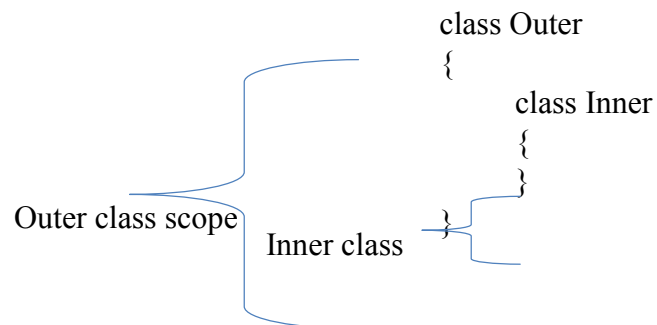
There are four types of inner classes as shown in the table 10.1.

Table 10.1: Types of Inner classes

Sr No	Type of inner classes	Description
1	Nested inner class	Class created inside the outer class
2	Method Local inner class	Class created inside the method of outer class
3	Static inner class	Static class inside the outer class
4	Anonymous inner class	Class created for implementing an interface or extending the class but it has no name. it is decided by the compiler

Let's see in details the types of inner classes

Class defined inside the other class is called nested class.
For example



Here Inner class can access the methods and data members of outer class but not a vice a versa. Following code demonstrate the same.

Program 10.1:

```
public class Outer
{
    int a=10;
    class Inner
    {
        int p=20;
        void show(String a1)
        {
            System.out.println(a);
        }
    }
    public void disp()
    {
        System.out.println("in disp "+new Inner().p);
    }

    public static void main(String ar[])
```

```
{  
  
    Outer o1=new Outer();  
  
    o1.disp();  
  
}  
}
```

Output:

in disp 20

10.3 METHOD LOCAL INNER CLASSES

This type of the class is written inside the method of outer class. The inner class can access the methods and data members defined in outer class but class's own member functions and data is not accessible outside. The program 10.2 demonstrates the method local inner class.

Program 10.2: Demonstration of method local inner class

```
public class Outer  
{  
  
    int outp=7;  
  
    void outerClassMethod()  
  
    {  
  
        System.out.println("inside outerMethod");  
  
        int p=9;  
  
        // Inner class is local to outerMethod()  
  
        class Inner  
        {  
  
            int inp=10;  
  
            void innerClassMethod()  
  
            {
```

```

System.out.println("inside innerMethod-->" + p);

System.out.println("inside innerMethod-->" + outp);

    }

}

    Inner y = new Inner();

y.innerClassMethod();

System.out.println("outside class-->" + y.inp);

}

public static void main(String[] args)

{

    Outer x = new Outer();

x.outerClassMethod();

}

}

```

Output:

```

inside outerMethod
inside innerMethod-->9
inside innerMethod-->7
outside class-->10

```

Here in this program class *Inner* is defined inside the method of *outerClassmethod*. Scope of the class is limited to the method. Inner class have an access to the method and data members of outer class but its own method and data members can not be accessible to outside method without the object of class.

10.4 ANONYMOUS INNER CLASSES

Anonymous inner class means no name is assigned to the class. Compiler at the runtime assigns the name to the anonymous class. Purpose of the anonymous class is to extend the abstract class or implements an interface without defining the child class explicitly. Following code demonstrate how to define anonymous class.

Program 10.3: Demonstration of anonymous class.

```
//abstract class definition
abstract class Greet {
    abstract void greetSomeone();
}

//extending abstract class to define its method
class Hello extends Greet {
    void greetSomeone() {
        System.out.println("DO greetings --from extended class ");
    }
}

public class HelloTest {
    void anonymousMethod()
    {
        //here use the anonymous class to define the abstract method.
        Greet g1= new Greet(){ void greetSomeone(){
        System.out.println("Anonymous greeting --from anonymous class");}};
        //invoke the abstract method
        g1.greetSomeone();
        System.out.println("Anonymous class name "+g1);
    }

    public static void main(String []ar) {
        HelloTest h1 =new HelloTest();
        System.out.println("Demo class name "+h1);
        Hello h=new Hello();
        System.out.println("Extended class name "+h);
        h.greetSomeone();
        h1.anonymousMethod();
    }
}
```

Output:

Demo class name HelloTest@6d06d69c

Extended class name Hello@7852e922

DO greeting --from extended class

Anonymous greeting --from anonymous class

Anonymous class name HelloTest\$1@4e25154f

Output shows that compiler had created the anonymous class with name HelloTest\$1

10.5 STATIC NESTED CLASS

A static class inside the other (non-static) class is called nested static class. It is known that static members are directly accessible with the class name and non-static members are not directly accessible inside the static method/class scope. Program 10.1 demonstrates the accession of the static variables.

Program 10.1: Example for accessing the static variable

```
public class staticdemo {
    static int sp=12;
    int nsp=24;
    public static void main(String ar[])
    {
        System.out.println("Static variable --> "+sp);
        System.out.println("Non-Static variable --> "+new staticdemo().nsp);
    }
}
```

Output:

Static variable --> 12

Non-Static variable --> 24

Here static variable is directly accessible in the static main method but not the same case for the non-static variable. It requires the object reference.

Following program 10.2, demonstrate the use of static nested class.

Program 10.2: Demo of static nested class

```
public class Outer
{
    static int data=30;
```

```
int p=10;

static class Inner
{
    int stat_p=20;
    void show()
    {
        System.out.println("accessing static data in static class "+ data);
        System.out.println("accessing non-static data in static class "+ new
        Outer().p);
    }
}

void disp()
{
    System.out.println("--accessing static class data in non-static class method
    --> "+ new Inner().stat_p );
}

public static void main(String args[])
{
    Outer.Innerobj=new Outer.Inner();
    obj.show();
    new Outer().disp();
}
}
```

Output:

accessing static data in static class 30

accessing non-static data in static class 10

--accessing static class data in non-static class method --> 20

10.6 SUMMARY

The chapter gives the brief introduction about the inner class. Inner classes restrict the use of data members and member function to the outside world. Anonymous classes ease the job of implementing the interface and

extending abstract class. One can use abstract class or interface without defining their implementation classes with the help of anonymous class.

10.7 EXERCISE:

1. What is nested class? Why there is a need of nested class?
2. What are the types of nested class?
3. What is the outcome of following program?

```
public class Excercise1
```

```
{  
    String s;  
    static class Inner  
    {  
        void innerMethod()  
        {  
            s = "First problem";  
        }  
    }  
}
```

4. What is a use of anonymous inner classes in java?

10.8 REFERENCES:

1. H. Schildt, *Java Complete Reference*, vol. 1, no. 1. 2014.
2. E. Balagurusamy, *Programming with Java*, Tata McGraw-Hill Education India, 2014
3. Sachin Malhotra & Saurabh Choudhary, *Programming in JAVA*, 2nd Ed, Oxford Press
4. The Java Tutorials: <http://docs.oracle.com/javase/tutorial/>



AWT

Unit Structure

- 11.0 Objective
- 11.1 Introduction,
- 11.2 Components,
- 11.3 Containers
- 11.4 Event-Delegation-Model and Listeners,
- 11.5 Button
- 11.6 Label
- 11.7 CheckBox, and CheckboxGroup
- 11.8 TextComponents: Text Field and Text Area
- 11.9 List
- 11.10 Choice
- 11.11 Menu
- 11.12 Layout Managers
 - 11.12.1 FlowLayout
 - 11.12.2 BorderLayout
 - 11.12.3 CardLayout
 - 11.12.4 GridLayout
 - 11.12.5 GridBagLayout
- 11.12 Summary
- 11.14 Exercise
- 11.15 References

11.0 OBJECTIVE

Objective of this chapter is

- To learn how to do GUI programming in Java.
- To understand the components and containers in Java.
- To understand the event Delegation Model in Java Programming
- To understand what is Layout and how to use various layouts in java.

11.1 INTRODUCTION

AWT

AWT is java's first User Interface.

AWT are called heavy weight components as they use the resources of underline operating system. AWT components will have different look and fill for the different platforms like windows, Linux, MAC OS etc. The hierarchy of AWT classes is shown in figure 11.1.

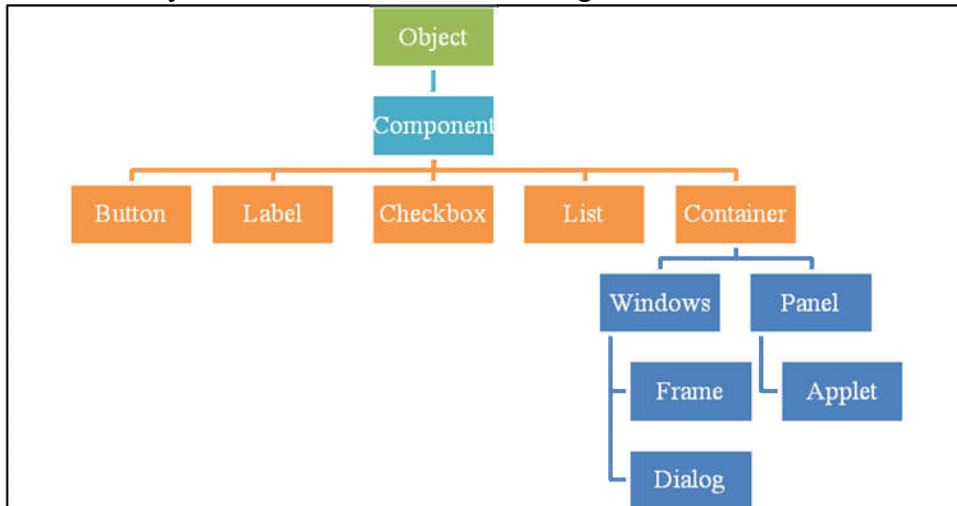


Figure 11.1.: Hierarchy of AWT

11.2 COMPONENTS

Component is an abstract superclass for all visual components in AWT (as shown in figure 11.1). Component is responsible for the remembering the background, foreground colour and the font. Table 11.1 shows some of the methods of Component class

Table 11.1 Methods of Component class

Methods	Description
public void setSize(int width,int height)	Set the size of the component with specific width and height
public void setLayout(LayoutManagemr)	Set the layout manager for the component.
public void setVisible(boolean status)	Set the visibility of control. If status is true. Control is visible otherwise not.
Public Graphics getGraphics()	Graphics context is obtained by calling getGraphics() method.
Void setBackground(Color c1)	Set the background color of the component

11.3 CONTAINER

Container is where components are added. To nest the component there is a requirement of container. To place the components at specific location, to group some components together, containers are used. As shown in figure 11.1, there are four components in AWT, window, frame, panel, dialog, and applet. Table 11.2 shows some common methods of Container class

Table 11.2: Methods of Container class

Methods	Description
Componentadd(Component comp)	Add the component c
void remove(Component c)	Removes the component c from container
Insets getInsets()	Insets is the amount of space leave in between the container and the window which contains the container
LayoutManager getLayout()	Get the layout manager of this container
void removeAll()	Removes all components from invoking container

Now let's see the types of the container.

11.3.1 Window:

Window is a top-level container which provides the display surface. It is not contained within any object. It has no border, title bar, menu bar. We can't create window's object directly. Instead, we use its subclasses Frame or Dialog.

11.3.2 Frame:

Frame is a subclass of a window and has border, menu bar, and title bar. Table 11.3 shows the methods and the constructors of Frame class.

Table 11.3 Methods of Frame class

Methods	Description
Frame()	Creates empty frame
Frame(String title)	Create a frame with title
Void setTitle(String title))	Sets the title for a frame window
Void setSize(int w, int h)	Set the width and height for a frame window
Void setVisible(boolean v)	Set the visibility for frame window. If v is true frame is visible otherwise not.

Program 11.1 is for frame window using association (creating object of frame class).

Program 11.1: Frame using Association

```
import java.awt.*;

public class framedemo{

    framedemo()  {

        Frame f=new Frame("Frame window Demo");

        f.setSize(600,500);

        f.setVisible(true);

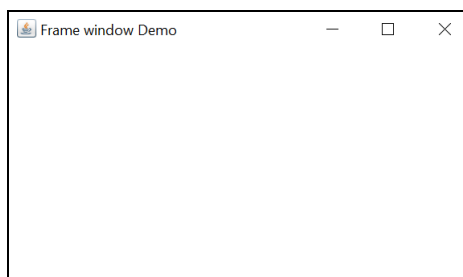
    }

    public static void main(String args[]){

        framedemo f=new framedemo();  }

}
```

Output:



Program 11.2 shows how to use Frame class using inheritance.

Program 11.2. Frame using Inheritance

```
import java.awt.*;

public class framedemo extends Frame{

    framedemo()  {

        setTitle("Frame using inheritance ");

        setSize(600,200);

        setVisible(true);

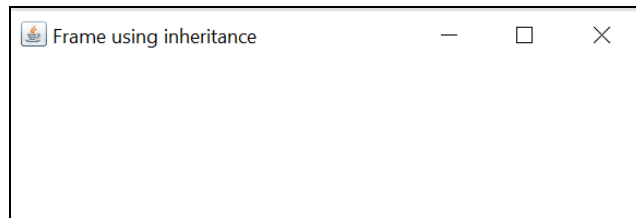
    }

    public static void main(String args[]) throws Exception

    {    framedemo f=new framedemo();  }

}
```

Output:



11.3.3 Dialog:

Dialog is a container which required a parent container. This is used for accepting inputs from user or to display the information to user. It will get close if parent window will close. Program 11.3 is the demonstration of how to use Dialog class.

Program 11.3: Dialog class Demo

```
import java.awt.*;

public class DialogDemo extends Frame {

    public DialogDemo() {

        setTitle("Frame Window");

        setSize(400,600);

        Dialog d = new Dialog(this, false);

        d.setLocation(100, 100);

        d.setTitle("Dialog Window");

        d.setSize(200, 200);

    }

}
```

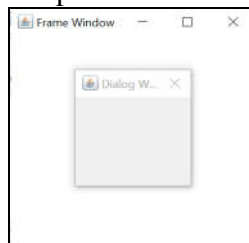
```

d.setVisible(true);
setVisible(true);
}

public static void main(String args[]) throws Exception
{
    DialogDemo f=new DialogDemo();
}

```

Output:



11.3.4 Panel:

Panel does not have borders. It is a simple container use for grouping the controls. Table 11.4. shows the constructors of Panel class.

Table 11.4: Constructors of Panel class

Methods	Description
Panel()	Creates the panel object with default layout manager
Panel(LayoutManager lm)	Creates a panel with specific layout manager

Example:

```

Panel p=new Panel();
p.add(new Button());

```

the above code creates the Panel as a window and contains component Button. Panel is not a main container. Now let's see how events are designed in AWT.

11.4 EVENT DELEGATION MODEL AND LISTENERS IN JAVA:

In GUI based programming, user communicates with the program by performing certain actions such as button click, key typing, closing and opening of window etc. This action causes the state change. Here we define the event as change in state of an object.

For example, we click on submit button present on the form. The form is submitted means the data gets saved on server and we received page or message saying your form is submitted successfully.

Here the source of the action is Button and its state change means button gets presses. This change in state causes some activity happen like form get submitted etc. This whole mechanism is called as Event Handling.

Java uses the event delegation model as shown in figure 11.2.

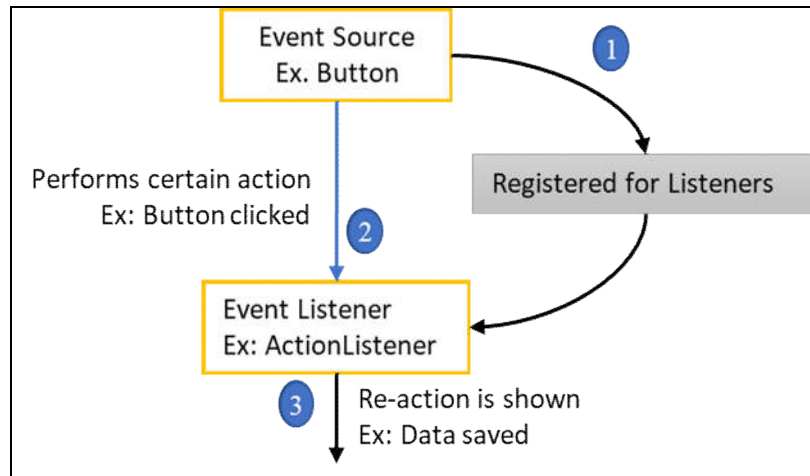


Figure 11.2: Event delegation model in java

Key components of event delegation model are shown in figure 11.2 with numbers.

1. Event Source: it is source which generates the events. Components such as Button, Frame, Textbox etc. are event sources.
2. Events: It is a change in state accurse in object
3. Listeners: They listen for the event which occurs. They get the notification of the event for which they are registered.

EventObject is the supper class for all the events defines in AWT. Table 11.5 shows the methods of EventObject class

Table 11.5: Methods of EventObject class

Method	Description
EventObject(Object source)	Constructs the Event object for the source object
Object getSource()	Returns the source object which regenerates the event
String toString()	Returns the string representation of the event

How to write the code for event handling?

AWT

Step1: Import java.awt.event.*

Step2: Implement an appropriate listener for the event

Implements the Listener for the event

Ex: public class abc extends Frame implements ActionListener {

Step3: Register the source for the event listener

this.addActionListener(this)

Step4: Implement the even handlers

public void actionPerformed(ActionEvent ae)

```
{  
    -----  
}
```

Table 11.6 shows the list event and respective event listeners for the various components.

Table 11.6: List of controls, Listeners and respective event class

Controls	Listeners	Event Handlers	Event Class	Trigger time
Button, List, MenuItem, TextField	Action Listener	Public void actionPerformed (Action Event ae)	ActionEvent	Button Pressed List Item double clicked Menu Item selected
Checkbox, Choice, List	ItemListener	void item State Changed (Item Event ie)	ItemEvent	Checkbox item or List item is clicked
Canvas, Dialog, Frame, Panel, Window	Mouse Listener	void mouse Pressed (Mouse Event me) void mouse Released (Mouse Event me) void mouse Entered (Mouse Event me) void mouseExited(M ouseEvent me)	MouseEvent	mouse is moved, mouse button is pressed or released, etc.

Dialog, Frame	Window- Listener	void windowClosing (Window Event we) void window Opened (Window Event we) void window Deiconified (Window Event we) void window Closed (Window Event we) void window Activated (Window Event we) void window Deactivated (Window Event we)	Window Event	window is activated, deactivated, window is closed or closing
Canvas, Dialog, Frame, Panel, Window	Mouse Motion- Listener	void mouse Dragged (Mouse Event me) void mouse Moved (Mouse Event me)	Mouse Event	mouse is dragged or moved
Component	Key Listener	void key Pressed (Key Event ke) void key Released (Key Event ke) void key Typed (Key Event ke)	Key Event	key is pressed, released and typed
Text- Component	TextListener	void text Changed (Text Event te)	TextEvent	Text is typed/entered in the textbox

Now let's see the various controls in AWT with the event they support.

11.5 BUTTON:

AWT

This is push button when pressed action is triggered. Used for creating navigational buttons, for submitting the form. Constructors and methods are shown in table 11.7

Table 11.7: Methods of Button class

Methods	Description
Button()	Creates the button object with no label on it
Button(String lbl)	Creates the button object with given label on it
void setLabel(String str)	Set the new label for the button
String getLabel()	Returns the label of the button on which this method is called
Void addActionListener(ActionEvent ae)	Register the button object for ActionListener
Void removeActionListener(ActionEvent ae)	Remove the ActionListener for the button object.

11.6 LABEL:

It's a simple component used to display a string. Constructors and methods are shown in table 11.8. Static fields defined in label class are

1. **static int LEFT:** the label is placed to left
2. **static int RIGHT:** the label is placed to right.
3. **static int CENTER:** the label is placed to centre.

Table 11.8: Methods of Label class

Methods	Description
Label()	Constructs the label with no caption
Label(String text)	Constructs the label with caption
Label(String text, int alignment)	Constructs the label with caption and aligned it to the left, right or centre as specified Ex: Label("Name",Label.CENTER)

void setText(String text)	Sets the caption/label to the label
String getText()	Returns the caption/label of the label
int getAlignment()	Returns the alignment value of the label
void setAlignment(int alignment)	Set the specified alignment value to the label

Program 11.4 demonstrates the use of Button and Label class

11.7 CHECKBOX AND CHECK BOX GROUP

Checkbox:

It is used to select the option as ‘on’ or ‘off’. When we select or deselect the checkbox, their states get changed and ItemEvent is fire. Table 11.9 shows the methods of Checkbox class.

Table 11.9: Methods of Checkbox class

Methods	Description
Checkbox()	Constructs the checkbox with no label/string
Checkbox(String label)	Constructs the checkbox with label/string
Checkbox(String label, Boolean state)	Constructs the checkbox with label/string and given state
Checkbox(String label, boolean state, CheckboxGroup chk)	Constructs the checkbox with label/string, its initial state and its specified checkbox group
Void addItemListener (ItemListener al)	It registers the checkbox for item listener.
Boolean getState()	Returns the state of the checkbox. True if it is selected otherwise false.

Check box Group

It is used to group the checkbox together. Once the group is created, only one checkbox among the given is selected (like radio button). Table 11.10 shows the methods of Check box Group class

Table 11.10: Methods of Check box Group class

AWT

Methods	Description
CheckboxGroup()	Create the instance of checkbox group
Checkbox getSelectedCheckbox()	Returns the selected checkbox from the group

Program 11.4 demonstrate the use of Checkbox components

11.8 TEXT COMPONENT:

It is a superclass for a TextField and TextArea Class that allows the user to enter the text.

Text Field and Text Area:

TextField creates a single line where as TextArea for multiline text. Table 11.11 and table 11.12 shows some of the methods of TextField and TextArea class

Table 11.11: Methods of Textfield class

Methods	Description
TextField()	Creates TextField component with no text
TextField(String text)	Creates TextField component with initial text
TextField(int n)	Creates TextField component with n number of columns
TextField(String text,int n)	Creates TextField component with initial text and with n columns.
void setText(String t)	Set the text t for the textfield

Table 11.12: Methods of TextArea class

Methods	Description
TextArea()	Creates TextArea component with no text
TextArea (String text)	Creates TextArea component with initial text
TextArea (int row, int column)	Creates TextArea component with rows and columns
TextArea (String text, int row, int column)	Creates TextArea component with initial text and with rows and columns.
TextArea (String text, int row, int column, int scrollbars)	Creates TextArea component with initial text and with rows and columns with visibility
void setText(String t)	Set the text t for the TextArea

Program 11.4: Program demonstrate the use of Button, Label, Checkbox, TextField

```
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
public class chkdemo implements ItemListener,ActionListener
{
    Label lbl1,lbl2,lbl3;
    Checkbox checkbox1,checkbox2,r1,r2,r3;
    TextField txtname,t2;
    Button submit;
    CheckboxGroup cbg;
    chkdemo()
    {
        Frame f = new Frame("Checkbox Example");
        f.setLayout(new GridLayout(5,2));
        checkbox1 = new Checkbox("C++");
        lbl1=new Label("Enter your Name");
        lbl2=new Label();
        lbl3=new Label();
        txtname=new TextField();
        submit=new Button("save");
        checkbox2 = new Checkbox("Java", true);
        cbg=new CheckboxGroup();
        r1=new Checkbox("8th",false, cbg);
        r2=new Checkbox("9th",false, cbg);
```

```

r3=new Checkbox("10th",false, cbg);

f.add(lbl1);

f.add(txtname);

f.add(checkbox1);

checkbox1.addItemListener(this);

checkbox2.addItemListener(this);

submit.addActionListener(this);

f.add(checkbox2);

f.add(r1);f.add(r2);f.add(r3);

f.add(lbl2);

f.add(submit);

f.add(lbl3);

f.setSize(400,400);

f.setVisible(true);

}

public void itemStateChanged(ItemEvent e)

{

    if(e.getSource()==checkbox1)

        lbl2.setText("C++: " +

(e.getStateChange()==1?"checked":"unchecked"));

    if(e.getSource()==checkbox2)

        lbl2.setText("Java : " +

(e.getStateChange()==1?"checked":"unchecked"));

}

public void actionPerformed(ActionEvent ae)

{

    if(ae.getSource()==submit)

    {

        lbl3.setText("Dear "+txtname.getText()+" Your data is saved ");

    }

}

```

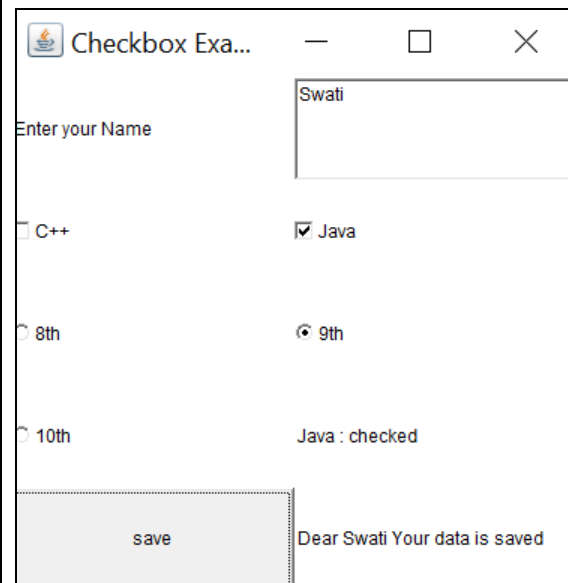


```

    }

    public static void main (String args[])
    {
        new chkdemo();
    }
}

```

Output:

11.9 LIST:

This component displays the list of text items. Here user can select one or multiple items from the list. Table 11.13 lists the methods of the List class.

Table 11.13: Methods of List class

Methods	Description
List()	Constructs the empty list
List(int num)	Constructs the list with number of lines specified visible.
List(int num, Boolean mode)	Constructs the list with number of lines visible and mode of selection. If true then list is with multiselects option otherwise false for single item select.
void add(String item)	Add the given item in the list

<code>void add(String item, int index)</code>	Add the given item at the given position in the list
<code>void deselect(int index)</code>	Deselects the given item in the list
<code>String getItem(int index)</code>	Returns the item present at specified index position
<code>int getItemCount()</code>	Returns the total item present in the list
<code>String[] getItems()</code>	Returns the names of the items present in the list
<code>int getRows()</code>	Returns the count of visible lines in the list
<code>int getSelectedIndex()</code>	Returns the index value of the selected item in the list
<code>VoidsetMultipleMode(boolean mode)</code>	Set the multiple selection mode for the list if value is true otherwise set single selects
<code>void remove(String item)</code>	Removes the specified item from the list
<code>void select(int index)</code>	Selects the item in the given index position

Following program 11.5, shows the demonstration of List class

Program 11.5: List Demo
<pre> import java.awt.*; import java.awt.event.ActionEvent; import java.awt.event.ActionListener; public class ListDemo implements ActionListener{ Label lbl; Button btnadd; List l1,l2; ListDemo() { Frame f = new Frame(); l1 = new List(5); l2 = new List(5); btnadd=new Button("Add"); </pre>

```

l1.add("Java Programming");

l1.add("C Programming");

l1.add("Python Programming");

l1.add("C++ Programming");

l1.add("C#");

btnadd.addActionListener(this);

f.add(l1);

f.add(l2);

f.add(btnadd);

f.setSize(400, 400);

f.setLayout(new FlowLayout());

f.setVisible(true);

}

public void actionPerformed(ActionEvent ae) {

    if(ae.getSource()==btnadd)

        l2.add(l1.getSelectedItemAt());

}

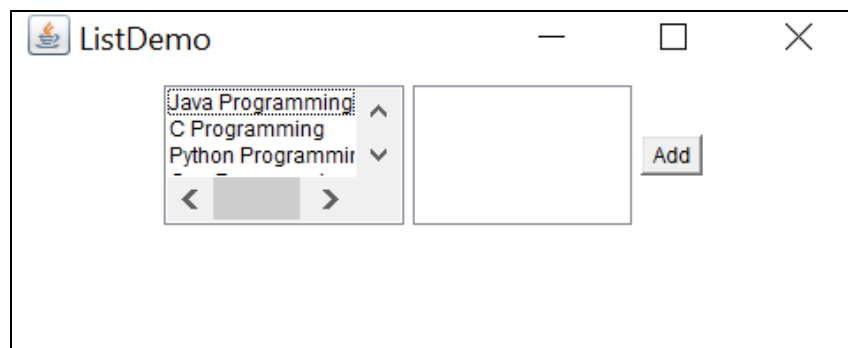
public static void main(String args[]) {

    new ListDemo();

}

}

```

Output:

11.10 CHOICE:

AWT

Choice is a drop-down list component. User can select any one item from the list. Every item in the choice has index value. Table 11.14 shows the methods of Choice class.

Table 11.14: Methods of Choice class

Methods	Description
Choice()	Constructs the empty choice list
void add(String item)	Adds the specified item in the choice
String getItem(int index)	Returns the string/item present in the given index position
int getItemCount()	Returns the number of items in the choice.
String getSelectedItem()	Returns the selected string/item.
int getSelectedIndex()	Returns the index value of the selected string/item.
void insert(String item, int index)	Insert the specific item at the specific given index position.
void remove(int position)	Removes the item from the given index position
void remove(String item)	Removes the specified item from the choice
void removeAll()	Remove all items from the choice.
void addItemListener(ItemListener l)	Register the choice component for the ItemListener
void removeItemListener(ItemListener l)	Remove the registration of choice component for the ItemListener
void select(int pos)	Selects the item present at specified index position
void select(String str)	Selects the specified item name

Program 11.6: Choice class demo

```
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
public class ChoiceDemo extends Frame implements ActionListener {
    Choice c;
    Button b1;
    Label lbl;

    ChoiceDemo() {
        // creating a choice component
        c = new Choice();
        b1=new Button("Show");
        lbl=new Label();
        setLayout(new GridLayout(2,2));
        c.add("Mumbai");
        c.add("Delhi");
        c.add("Chennai");
        c.add("Jaipur");
        c.add("Banglore");
        b1.addActionListener(this);
        add(c);
        add(b1);
        add(lbl);
        setSize(200, 200);
        setVisible(true);
    }

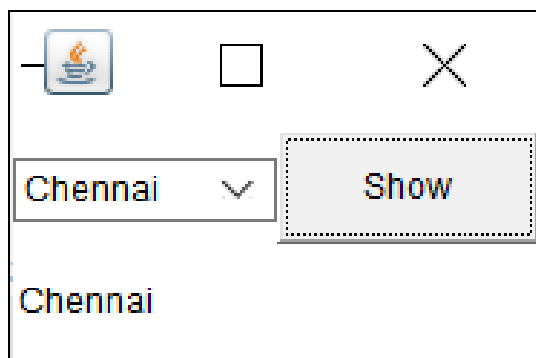
    public void actionPerformed(ActionEvent ae)
    {
        if(ae.getSource()==b1)
            lbl.setText(c.getSelectedItem());
    }

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

```

new ChoiceDemo();
}
}

```

Output:

11.11 MENU:

Menu is the list of pop up items associated with top level windows. AWT provides three classes MenuBar, Menu, and MenuItem. MenuBar has multiple Menus and each menu can have sub-menus in drop-down list form. Here MenuItem is the superclass of Menu. CheckboxMenuItem will create the checkable menu item. Table 11.15 shows the methods of Menu class.

Table 11.15: Methods of Menu class

Methods	Description
Menu()	Construct a new menu with no label
Menu(String label)	Construct a new menu with label
MenuItem add(MenuItem mi)	Add the menu item to the menu
void add(String label)	Add the item with given label
void addSeparator()	Add separator between menu
int countItems()	Returns the items in the menu
void insert(MenuItem menuitem, int index)	Insert the menu item at specific given index position
void remove(int index)	Removes the item present at given index position

Program 11.7: Demonstrate the MenuBar, MenuItem, Menu class

```
import java.awt.*;

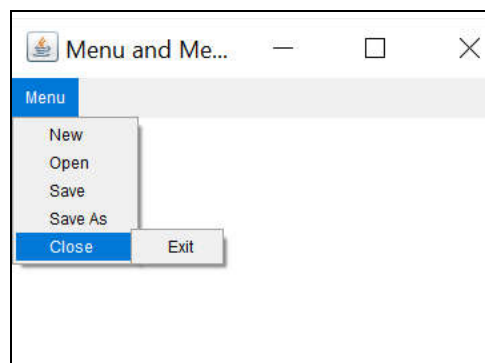
public class MenuDemo extends Frame
{
    MenuDemo(){
        setTitle("Menu and MenuItem Example");
        MenuBar mb=new MenuBar();
        Menu menu=new Menu("Menu");
        Menu submenu=new Menu("Close");
        MenuItem i1=new MenuItem("New");
        MenuItem i2=new MenuItem("Open");
        MenuItem i3=new MenuItem("Save");
        MenuItem i4=new MenuItem("Save As");
        MenuItem i5=new MenuItem("Exit");
        menu.add(i1);
        menu.add(i2);
        menu.add(i3);
        menu.add(i4);
        submenu.add(i5);
        menu.add(submenu);
        mb.add(menu);
        setMenuBar(mb);
        setSize(400,400);
        setLayout(null);
        setVisible(true);
    }

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

```

{
new MenuDemo();
}
}

```

Output:

11.12. LAYOUT MANAGER:

When the GUI is designed, the components are placed at some defined location using `setBounds(int x, int y, int w, int h)` or by using `setSize(int w, int h)` and `setLocation(int x, int y)` method. If numbers of components are more, it becomes difficult to define the position and size of each component. AWT supports predefined layout manager classes which helps to place the components on the define container. AWT supports following Layout Managers namely:

- FlowLayout
- BorderLayout
- CardLayout
- GridLayout
- GridBagLayout

`setLayout(LayoutManager obj)` method is used to set a layout to a container. Let's see one by one these layout managers.

11.12.1 FlowLayout:

FlowLayout places the components in flow manner i.e. one after another in a line starting from left to right and top to bottom. Table 11.16 shows the constructors and methods of FlowLayout. Program 11.8 shows how to use of FlowLayout.

Table 11.16: Methods of FlowLayout class

Constructors/Methods	Description
FlowLayout()	Creates a default FlowLayout which place the components starting from centre of first line. By default, the space between two components is 5 pixel.
FlowLayout(int alignment)	Creates a FlowLayout with 5 pixel space between each components and places the component as per the alignment specified FlowLayout.LEFT FlowLayout.RIGHT FlowLayout.CENTER FlowLayout.LEADING FlowLayout.TRAILING
FlowLayout(int alignment, int hgap, int vgap)	Creates FlowLayout with given alignment and the spacing mentioned

Program 11.8: FlowLayout Demo

```
import java.awt.*;

public class FlowLayoutDemo {
    Frame f;

    FlowLayoutDemo()
    {
        f = new Frame();

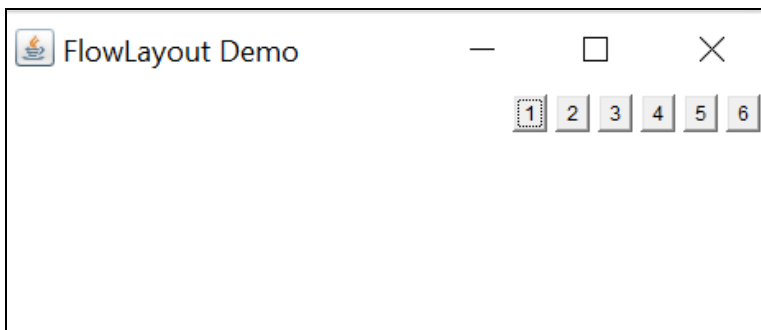
        Button b1 = new Button("1");
        Button b2 = new Button("2");
        Button b3 = new Button("3");
        Button b4 = new Button("4");
        Button b5 = new Button("5");
        Button b6 = new Button("6");

        f.add(b1);
        f.add(b2);
```

```
f.add(b3);  
f.add(b4);  
f.add(b5);  
f.add(b6);  
  
f.setLayout(new FlowLayout(FlowLayout.RIGHT));  
f.setSize(300, 300);  
f.setVisible(true);  
}  
  
public static void main(String argsv[])  
{  
    new FlowLayoutDemo();  
}  
}
```

AWT

Output:



11.12.2 BorderLayout:

BorderLayout arrange the components in five different regions, namely centre, east, west, north, and south. Each region holds only one component.

Following constants represents the region,

- BorderLayout.NORTH
- BorderLayout.SOUTH
- BorderLayout.EAST
- BorderLayout.WEST
- BorderLayout.CENTRE

A component is explicitly added to the one of the above said region using method,

Add(Component c, Object region)

Ex: to add a button in south region use

add (new Button(b1), BorderLayout.SOUTH)

Table 11.17 shows the constructors and methods of BorderLayout and program 11.9 shows how to use BorderLayout.

Table 11.17: Methods of BorderLayout class

Constructors/Methods	Description
BorderLayout()	Creates a default BorderLayout which place the components at centre.
BorderLayout (int hgap, int vgap)	Creates a BorderLayout with specific space/gaps between components.

Program 11.9 BorderLayout Demo

```
import java.awt.BorderLayout;
import java.awt.Button;
import java.awt.FlowLayout;
import java.awt.Frame;

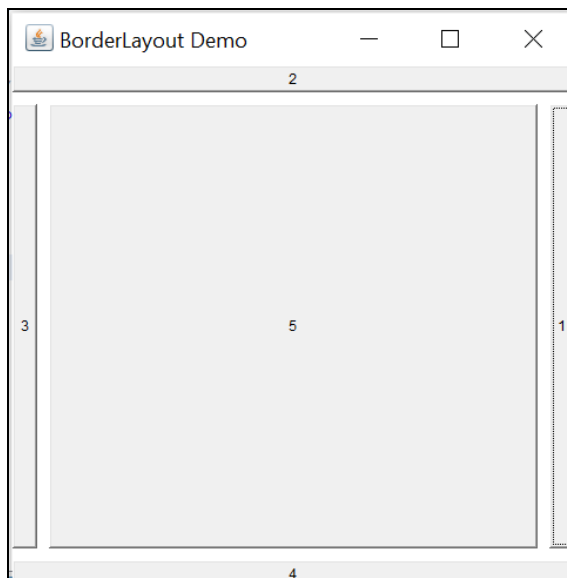
public class BorderLayoutDemo extends Frame
{
    BorderLayoutDemo()
    {
        setLayout(new BorderLayout(10,10));
        setSize(500, 500);
        setTitle("BorderLayout Demo");

        Button b1 = new Button("1");
        Button b2 = new Button("2");
        Button b3 = new Button("3");
        Button b4 = new Button("4");
        Button b5 = new Button("5");
```

```
add(b1,BorderLayout.EAST);  
add(b2,BorderLayout.NORTH);  
add(b3,BorderLayout.WEST);  
add(b4,BorderLayout.SOUTH);  
add(b5,BorderLayout.CENTER);  
setVisible(true);  
}  
  
public static void main(String argsv[])  
{  
    new BorderLayoutDemo();  
}  
}
```

AWT

Output:



11.12.3 CardLayout:

CardLayout keeps the components like the cards i. e. components are stack and only one component is visible at a time. Table 11.18 shows the constructors and methods of CardLayout.

Table 11.18: Methods of CardLayout class

Constructors/Methods	Description
CardLayout()	Creates a default CardLayout
CardLayout (int hgap, int vgap)	Creates a CardLayout with specific space/gaps between components.
void first(Container deck)	Here deck is the parent container which holds the cards. First card in the deck is return
void last(Container deck)	Shows the last card on the container
void next(Container deck)	Shows the next card (in sequence) on the container
void previous(Container deck)	Shows the previous card (in sequence) on the container
void show(Container deck, String cardName)	Shows the specific given card on the container

Program 11.10 demonstrates the use of Card Layout.

Program 11.10 CardLayout Demo

```

import java.awt.BorderLayout;

Import java.awt.*;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class CardLayoutDemo extends Frame implements ActionListener{

    CardLayout crd;

    Panel cardp,nevigat;

    Button b1,b2,b3,b4,b5,first, last, next,previous,show;

    CardLayoutDemo() {

        //Set Layout for Main frame

        setLayout(new BorderLayout());

        setSize(500, 500);

        setTitle("CardLayout Demo");
    }

```

```
// set the cardlayout for first panel
cardp=new Panel();
crd=new CardLayout();
cardp.setLayout(crd);
b1 = new Button("1");
b2 = new Button("2");
b3 = new Button("3");
b4 = new Button("4");
b5 = new Button("5");
cardp.add("Button1",b1);
cardp.add("Button2",b2);
cardp.add("Button3", b3);
cardp.add("Button4",b4);
cardp.add("Button5",b5);
add(cardp,BorderLayout.CENTER);

// create a second panel; add navigation button ; set the flowlayout
Panel nevigat=new Panel();
nevigat.setLayout(new FlowLayout());
first = new Button("first");
last = new Button("last");
next = new Button("next");
previous = new Button("previous");
show = new Button("show");

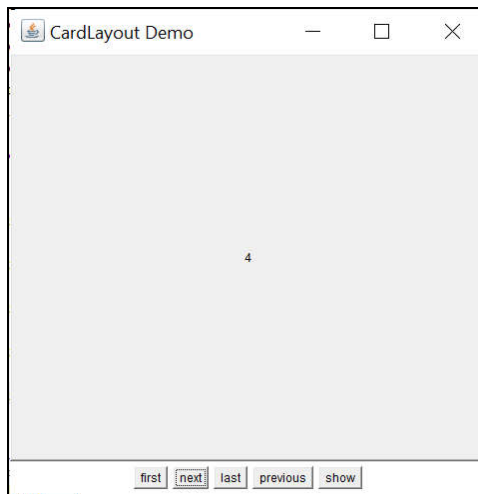
//register the navigation buttons for ActionListener
first.addActionListener(this);
last.addActionListener(this);
next.addActionListener(this);
previous.addActionListener(this);
```

```
        show.addActionListener(this);

        nevigat.add(first);
        nevigat.add(next);
        nevigat.add(last);
        nevigat.add(previous);
        nevigat.add(show);
        add(nevigat, BorderLayout.SOUTH);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent ae)
    {
        // on click of respective navigation button, card is displayed
        if(ae.getSource()==last)
            crd.last(cardp);
        if(ae.getSource()==first)
            crd.first(cardp);
        if(ae.getSource()==next)
            crd.next(cardp);
        if(ae.getSource()==previous)
            crd.previous(cardp);
        if(ae.getSource()==show)
            crd.show(cardp, "Button3");
    }

    public static void main(String argsv[])
    {
        new CardLayoutDemo();
    }
}
```



11.12.4 GridLayout:

This layout arranges the components in grid format i. e. in 2 X 2 matrixes. Table 11.x shows the constructors of GridLayout class. Table 11.19 shows the constructors and methods of GridLayout.

Table 11.19 Constructors of Grid Layout

Constructor	Description
GridLayout()	Creates the grid layout of single column
GridLayout(int r, int c)	Creates the grid layout of given rows and columns
GridLayout(int r, int c, int h_gap, int v_gap)	Creates the grid layout of given rows and columns and with specified gaps.

The program 11.11 demonstrate the use of GridLayout class

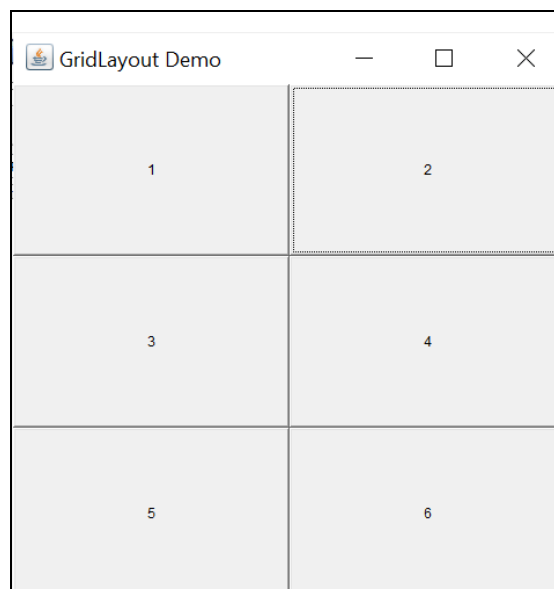
Program 11.11: Demonstration of GridLayout class

```
import java.awt.BorderLayout;
import java.awt.*;

public class GridLayoutDemo extends Frame {
    Button b1,b2,b3,b4,b5,b6;
    GridLayoutDemo()
    {
        setLayout(new GridLayout(3,2));
        setSize(500, 500);
    }
}
```



```
setTitle("GridLayout Demo");  
  
b1 = new Button("1");  
b2 = new Button("2");  
b3 = new Button("3");  
b4 = new Button("4");  
b5 = new Button("5");  
b6 = new Button("6");  
add(b1);  
add(b2);  
add(b3);  
add(b4);  
add(b5);  
add(b6);  
setVisible(true);  
}  
public static void main(String argsv[])  
{  
    new GridLayoutDemo();  
}  
}
```

Output:

11.12.5 Grid Bag Layout:

AWT

Grid Layout places the components in a grid in a sequence of adding those on window. All those components have same/equal fixed dimensions. Components' size can not be resized.

Gridbag Layout allows placing the components in a grid at any specified row and column with different (more than one cell) width and height i.e. component may have width of more than one column span and height of more than one row span. Table 11.20 shows the constructors and methods of Grid Bag Layout.

Table 11.20 Methods of GridBagLayout class

Methods	Description
GridBagLayout()	Creates a default GridBagLayout c
void setConstraints(Component comp, GridBagConstraints cons)	This method sets the constraint on the components which is to be placed on container. Here GridBagConstraints is a helper class which is used to set the constraints for components.

Table 11.21 describes the grid Bag Constraints' field and their purpose

Table 11.21 : Fields of Grid Bag Constraints

Methods	Description
int anchor	Specifies the location of a component within a cell. The default is

Following Program 11.12 demonstrate the use of GridBagLayout class.

Program 11.12: Demo of GridBagLayout class

```
import java.awt.*;  
import java.awt.Button;  
import java.awt.GridBagConstraints;  
import java.awt.GridBagLayout;  
public class GridBagLayoutDemo extends Frame  
{
```

```
public GridBagLayoutDemo()
{
    GridBagLayout gb = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();
    setLayout(gb);
    setTitle("GridBag Layout Example");
    //GridBagLayout layout = new GridBagLayout();
    //this.setLayout(layout);
    //constraints for textfield
    gbc.fill = GridBagConstraints.BOTH;
    gbc.gridx = 0;
    gbc.gridy = 0;
    gbc.gridheight=1;
    gbc.gridwidth=3;
    this.add(new TextField("Enter Number"), gbc);
    //constraints for button 1
    gbc.fill = GridBagConstraints.BOTH;
    gbc.gridx = 0;
    gbc.gridy = 1;
    gbc.gridheight=1;
    gbc.gridwidth=1;
    this.add(new Button("1"), gbc);
    //constraints for button 2
    gbc.gridx = 1;
    gbc.gridy = 1;
    gbc.gridheight=1;
    gbc.gridwidth=1;
    this.add(new Button("2"), gbc);
    //constraints for button 3
```

```
gbc.fill = GridBagConstraints.BOTH;

//gbc.ipady = 20;

gbc.gridx = 0;

gbc.gridy = 2;

gbc.gridheight=1;

gbc.gridwidth=1;

this.add(new Button("3"), gbc);

//constraints for button 4

gbc.gridx = 1;

gbc.gridy = 2;

gbc.gridheight=1;

gbc.gridwidth=1;

this.add(new Button("4"), gbc);

//constraints for button 2

gbc.gridx = 2;

gbc.gridy = 1;

gbc.fill = GridBagConstraints.BOTH;

gbc.gridwidth = 1;

gbc.gridheight=2;

this.add(new Button("+"), gbc);

gbc.gridx = 0;

gbc.gridy = 3;

gbc.fill = GridBagConstraints.BOTH;

gbc.gridwidth = 1;

gbc.gridheight=1;

this.add(new Button("="), gbc);

gbc.gridx = 1;

gbc.gridy = 3;

gbc.fill = GridBagConstraints.BOTH;
```

```

gbc.gridwidth = 2;

gbc.gridheight=1;

this.add(new Button("-"), gbc);

setSize(600, 600);

setPreferredSize(getSize());

setVisible(true);

}

public static void main(String[] args)

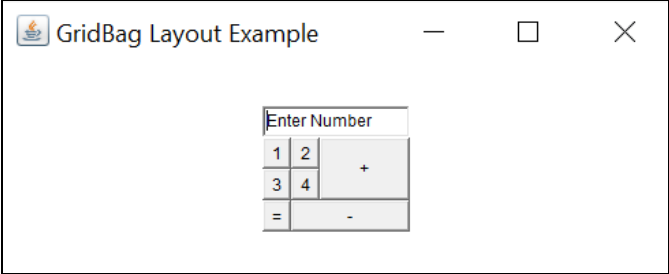
{

    GridBagLayoutDemo a = new GridBagLayoutDemo();

}

}

```



11.13 SUMMARY

- AWT package provides the different API for designing a user interface.
- Java supports the different kind of windows using Window, Frame, Dialog, Panel etc.
- Events are handling using different interfaces.
- Event handling is done by registering the event first, then implementing the event handler for the event.
- Java also defines the different layout managers for arranging the components on window.
- FlowLayout is default layout for panel and applet, BorderLayout is default layout for Frame.

11.14 EXERCISE:

AWT

1. Why AWT components are called as light weight components?
2. What is a difference between container and components?
3. How event delegationmodel in java?
4. What is Listener? How do any components respond to event?
5. What is the difference between TextArea and TextField?
6. What is the difference between Choice and List?
7. What is the difference between the Frame and Panel?
8. What is the use of Layout managers? Explain the difference between GridLayout and GridBagLayout manager class.

11.15 REFERENCES

1. H. Schildt, *Java Complete Reference*, vol. 1, no. 1. 2014.
2. E. Balagurusamy, *Programming with Java*, Tata McGraw-Hill Education India, 2014
3. Sachin Malhotra & Saurabh Choudhary, *Programming in JAVA*, 2nd Ed, Oxford Press
4. The Java Tutorials: <http://docs.oracle.com/javase/tutorial/>

