



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.

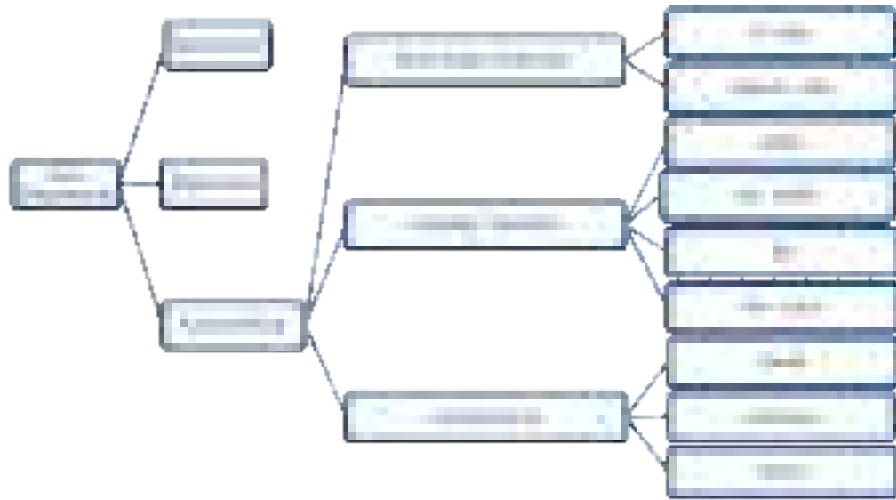
4. Operators: Operators are the symbols that performs operations. Java contains different types of operators like Arithmetic Operator (+, -, *, /, %), Logical Operator (&&, ||, ~,), Relational Operator(<, <=, >, >=, !=, ==), Bitwise Operators (&, |, ~), Shift Operators (<<, >>, >>>) , Assignment Operators(=,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

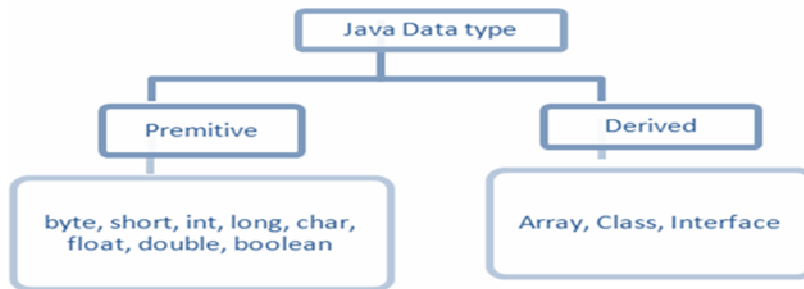
Java supports three jump statements: `break`, `continue`, and `return`. These statements transfer control to another part of a java program.

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**.

• Widening Casting(Implicit)



• Narrowing Casting(Explicitly done)



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[];`

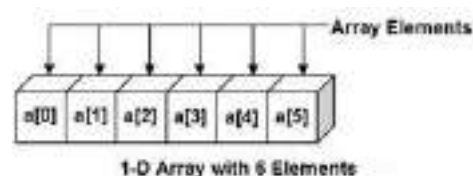
or `[access_specifier] type[] var-name;`

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.



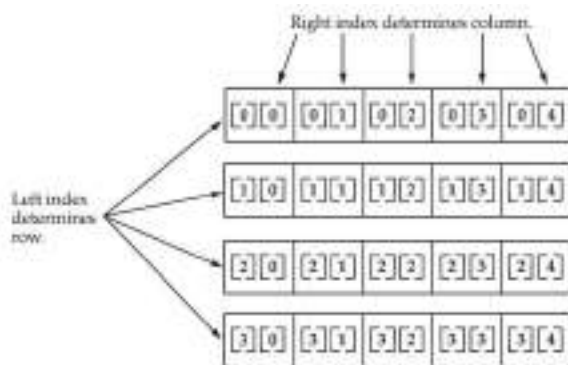
Declaration and initialization of array can be combined eg. `public float temp_jan[] = new float[31];`

A *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]`

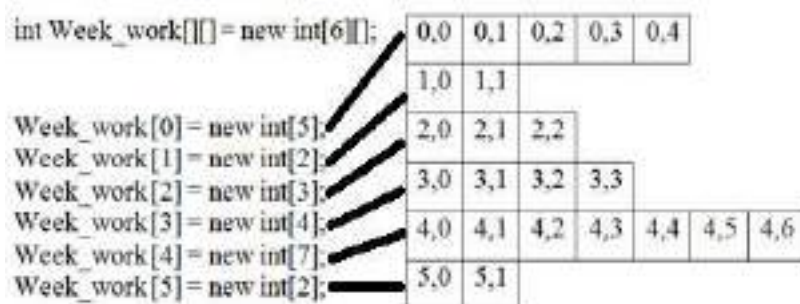
This will declare a 2D array of int to hold 3 rows and 3 columns.



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 & Saurabh Choudhary, 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.

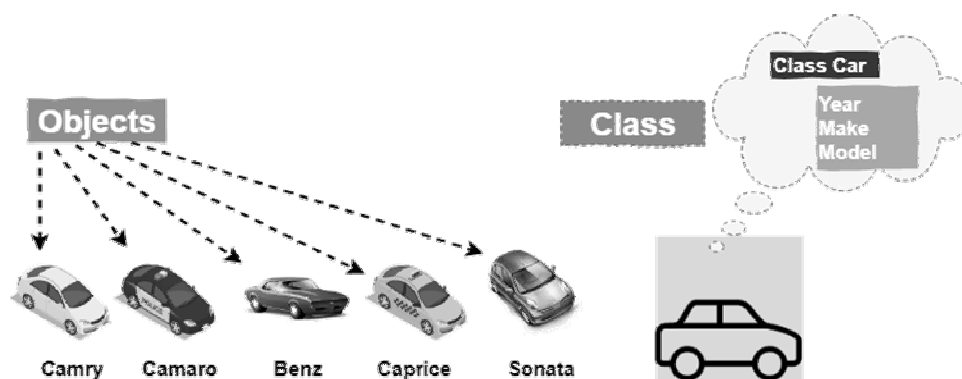
Encapsulation is the mechanism of binding together code and the data it manipulates, and keeps both safe from outside interference and misuse. It is like a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code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

```
}
```


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

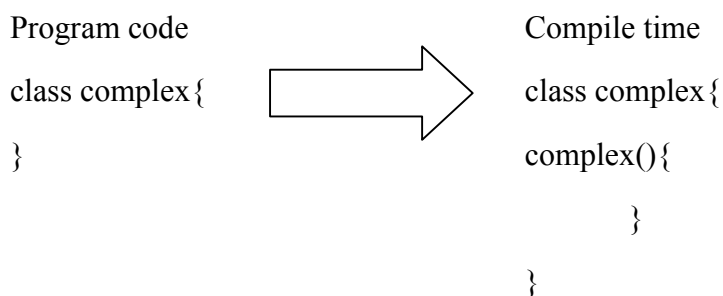
A constructor initializes an object at the time of creation. It has the same name as the class in which it resides. The constructor is automatically called when the object is created, before the `new` operator completes. Constructors have no return type, not even `void`. This is because the implicit 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.

```

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.



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.

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 (Package Inheritance)	Yes (Package)	No
From a subclass outside the same package	Yes	Yes (Inheritance)	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

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

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);

    }
}
```

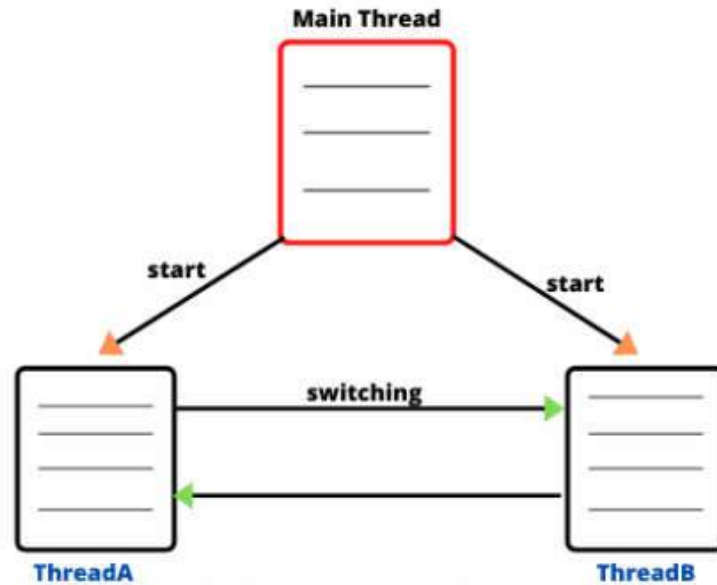


Fig. 5.2 A Multithreaded program

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);  
        }  
    }  
}
```

```
{  
  
    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();  
    }  
}
```

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);
        }
    }
}
```

```

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
{

```

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

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.

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.

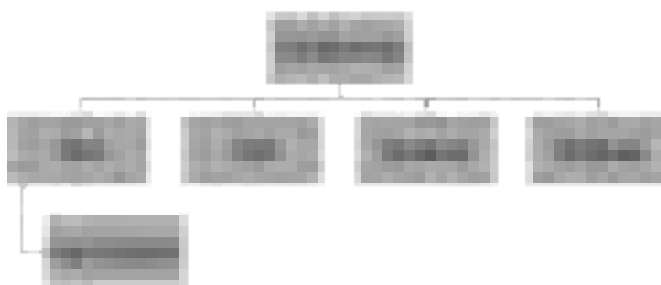


Figure 9.1: Collection Interfaces

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

```

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.

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.

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.

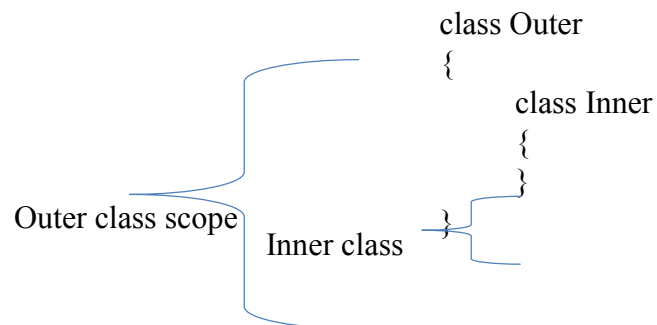
There are four types of inner classes as shown in the table 10.1.

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[])
```

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;
```

AWT is java's first User Interface.

AWT are called heavy weight components as they use the resources of underlying operating system. AWT components will have different look and feel 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 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

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.

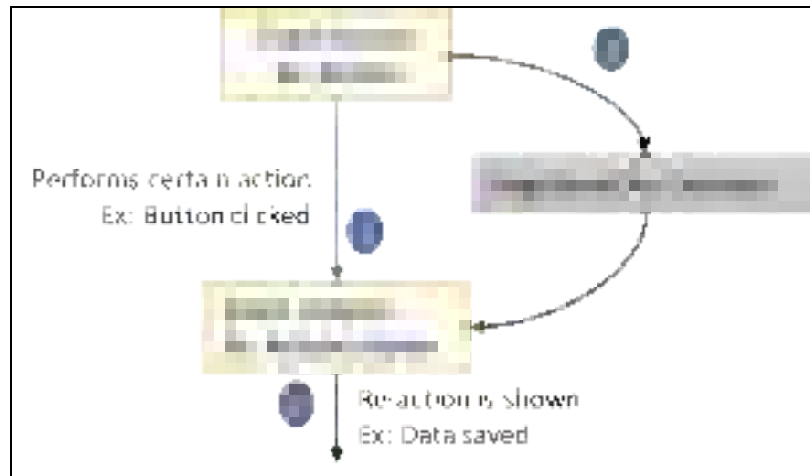


Figure 11.2: Event delegation model in java

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 occur 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 super 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)

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());
    }

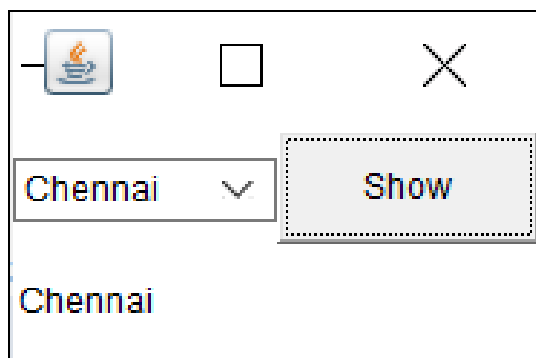
    public static void main(String args[])
    {
```

```

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.*;

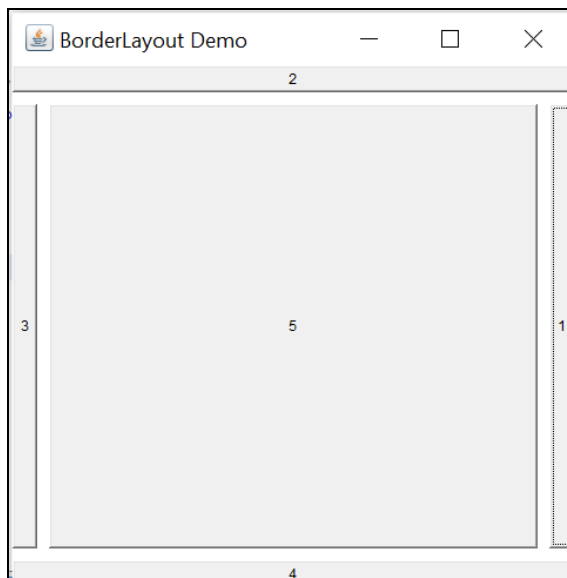
public class MenuDemo extends Frame
{
    MenuDemo(){
        setTitle("Menu and MenuItem Example");
        MenuBar mb=new MenuBar();
        Menu menu=new Menu("Menu");
        Menu submenu=new Menu("Close");
        MenuItem i1=new MenuItem("New");
        MenuItem i2=new MenuItem("Open");
        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[])
```

```
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");
    }

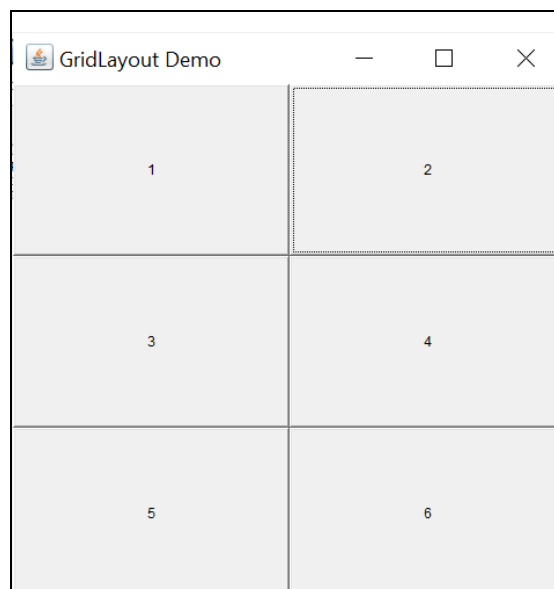
```

```
// 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);
```

```
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)

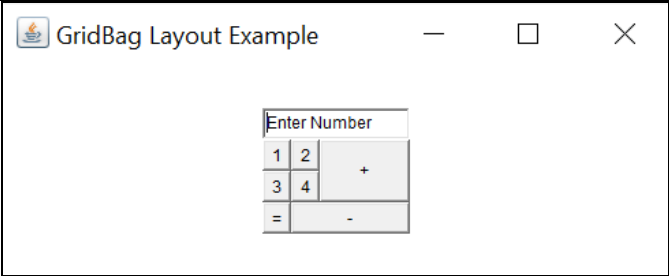
{

    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/>

